memory. After this is done, the resultıng configuration is said to use *vertical microprogramming*.

As an example of how the number of outputs might be reduced, consider that in some cases when one control signal is raised, another is always raised, and so these two signals could be combined into a single signal.

In some cases different control signals are never turned on at the same time. If N such signals can be found, then only M control lines, where $2^M > N$, will be required, and a decoder can be used to provide the necessary control signals. For instance, the ADD, SUB, RESET AC, and MB INTO BR signals are never turned on at the same time. Thus two control output lines with a four-output decoder could be used to generate these signals.

When vertical microprogramming is used, the system becomes less flexible, since if a microprogram is to be changed or enlarged, fewer options in control signal generation will be available. As a result most commercial computers are arranged so they are somewhere between horizontal and vertical in their construction. (Many schemes have been used, and several are described in the Questions.)

Microprogramming is widely used in the new computer lines. Since the instruction repertoire for the computer is effectively stored in the ROM, the instructions provided can be changed or added to by changing or adding to the ROM.

Further, microprogramming is useful in simulating one computer on another. Suppose that we have a computer which has a basic set of registers and operations between registers, and we have the ability to microprogram this computer. Further, we have a second computer with a certain set of instructions and a set of programs written to run on this second computer. We now wish to make the first computer run these programs and deliver the same results as the second computer would have delivered. This is called *simulation,* and the first computer is said to *simulate* the second computer. To do this, we microprogram the simulator computer so that a given instruction has the same effect as the same instruction in the second machine.[10]

As can be seen, a computer which is microprogrammed can be made to simulate another computer. Clearly some computers have architectures which are much better suited for simulation than others.

The microprograms provided by a manufacturer (or anyone else) to be used on its microprogrammed computer are generally called *firmware*. The instructions that a microprogrammed computer provides can be very complex and can be carefully designed to satisfy the programmer's needs. The primary objections to microprogramming are (1) speed, because the logic gates used in a "conventional" computer will be faster than the ROM in most cases, and so the conventional machine may run faster; (2) the gates can be minimized in number since the instructions are to be fixed, and thus the total amount of equipment can often be made smaller. (This is not always the case; however, ROMs are quite compact and inexpensive so that the advantage of gates decreases as time passes.) As a result, most large, fast "super" computers tend to use logic gates for control, while the medium and smaller computers now tend to be microprogrammed.

[10] This is often called *emulation* when microprogramming is used, it is necessary to rename registers and arrange for other changes to really effect this, but the principle is essentially given here.

SUMMARY

9.11 The instruction word formats and instruction repertoires for two single-address computers were discussed. A design for a single-accumulator computer's control section was then presented. This design is based on register operations which can be described by using register transfer language. The most used general procedures for control design and computer description were presented including timing operation and the use of tables of operations to implement instructions.

When a computer is microprogrammed, a ROM is used to store the control signals needed to sequence register operations. The preparation of the sequence of operations to be performed by the computer is called microprogramming and is often performed by using a register transfer language to describe these register microoperations.

If each control signal has a bit in the ROM output word dedicated to it, the microprogramming is said to be horizontal. When the size of the ROM is reduced by encoding ROM outputs and then decoding them by gates, the microprogramming is said to be vertical.

QUESTIONS

9.1 A single-address, one-instruction-per-word computer has a word length of 22 binary digits. The computer can perform 32 different instructions, and it has three index registers. The inner memory is a 16,000-word magnetic core memory. Draw a diagram of the computer word, allocating space for each part of the basic instruction word (OP-code part, address part, index register part). Do not use the sign digit (leftmost digit) of the word.

9.2 Design a single stage of an accumulator and *B* register which will add and shift left in one operation (step) or will simply shift left in one step. Use SHIFT LEFT and ADD AND SHIFT LEFT as control signals, a full-adder, AND and OR gates, and *RS* flip-flops.

9.3 Make out a timing table and modify the control circuitry in Fig. 9.6, including the modification in Fig. 9.8, so that the machine has an unconditional BRANCH instruction BRA, as well as a conditional BRANCH instruction BRM, generating the necessary control signals.

9.4 Discuss how you would expand or perhaps improve the register transfer language in Table 9.5. Do you think that microprogramming in a higher-level language, such as Pascal or one of its variations, would yield an efficient microprogram in the control memory? Discuss this.

9.5 Show how to modify Fig. 9.8 so that the BRM instruction becomes a BRP instruction, meaning that the computer jumps or branches when the ACC is positive instead of negative.

9.6 Show how to generate timing signals (such as T_0, T_1, T_2, and T_3 in Fig. 9.5) by using a shift register with the rightmost stages' outputs connected to the leftmost stages' inputs. This is called a *ring counter*. In what states would you set the flip-flops to start?

9.7 Why is the instruction counter always placed in the memory address register at time T_3 during the execution part of an instruction in Table 9.2?

9.8 Explain why some instructions require both execution and instruction cycles and others require only instruction cycles. Give examples of both kinds of instruction.

9.9 Why can the SET W and AC INTO MB control signals be combined in Fig. 9.6?

9.10 Why can the IC INTO MA, SET I, and CLEAR E control signals be combined in Fig. 9.6?

9.11 Show how to add a pushbutton connected to a reset and start wire which will (using DC SETs and DC RESETs on the flip-flops) cause the computer in Fig. 9.6 to start executing a program beginning at location 0 in the memory when it is depressed.

9.12 Add a BRANCH ON ZERO instruction similar to that in Table 9.3 and Fig. 9.8, except that the computer branches when the accumulator value is all 0s.

9.13 Add a SHIFT LEFT instruction to the example computer control, using a technique as shown in Table 9.4 and Figs. 9.8 and 9.9.

9.14 When microprogramming is used to generate control signals and a ROM is used, how can the instruction repertoire of the computer be changed?

9.15 In Fig. 9.10 the control signals could be loaded into *D* flip-flops and the flip-flops' outputs used as the actual control signals. Give advantages and disadvantages of this arrangement.

9.16 Explain the difference between a microoperation and the control signal which implements it.

9.17 To implement BRANCH or JUMP ON ACCUMULATOR NEGATIVE instructions, another control signal C_{27} can be added which, when it is a 1, causes a test of the sign bit of the accumulator and a jump in the control memory to a section of microoperations which cause the desired change in the computer sequence of operations. Design this instruction.

9.18 In writing microprograms it is convenient to have an IF microoperation. For instance, to write a microprogram to implement a branch instruction, we might like an IF ($AC_0 = 1$) THEN $C_{0-6} \rightarrow$ IAR ELSE IAR + 1 → IAR microoperation. This says if AC_0 is a 1, then place the current value of C_0 to C_6 in IAR, which means that the next microinstruction will be from the address given in C_0 to C_6. If AC_0 is a 0, the next microinstruction will be from the next location in the control store. Write a microprogram for the branch-register-minus (BRM) instruction, using this microoperation.

9.19 Write a microprogram for a BRP (branch or positive) instruction, using the information in Question 9.18.

9.20 Show how to implement the instruction in Question 9.19.

9.21 Write a microprogram for a SHIFT RIGHT instruction, using the IF type of statement just described. (You will also need a counter.)

9.22 Show how to implement the microprogram in Question 9.21.

9.23 Write a microprogram to implement a multiplication instruction.

9.24 Show how to implement the multiplication instruction in Question 9.23.

9.25 Write a microprogram to implement a DIVIDE instruction.

9.26 Show how to implement your DIVIDE instruction from Question 9.25.

9.27 Reduce the number of control signals used in Fig. 9.11.

9.28 Explain what features you might like in a computer, which is to be microprogrammed to simulate several other computers.

9.29 Discuss some of the advantages and disadvantages of microprogramming.

9.30 Some computers now use a branch or jump scheme where status bits stored in flip-flops are continually being set during arithmetic and logic operations. For instance, status bits Z and N are commonly used to indicate if the result of an operation is "all zero" or "negative." Show how to add such status bits to the arithmetic section of the computer shown in Fig. 9.6.

9.31 When status bits are used, jump instructions are of the form "jump on zero," meaning jump if the Z flip-flop is a 1, or "jump negative," meaning jump if the N flip-flop is a 1. Design these two instructions, using the Z and N circuitry from Question 9.30.

9.32 Discuss the advantages and disadvantages of *random logic* (gate-generated logic) versus microprogramming for a computer control section. Assume that the computer is a minicomputer.

9.33 Compare the microprogramming and conventional random logic techniques for generating the control signals in a general-purpose digital computer. Assume that the computer is to be sold in a large market where both business and scientific programs are to be run. Give the advantages and disadvantages of both techniques for implementing control logic.

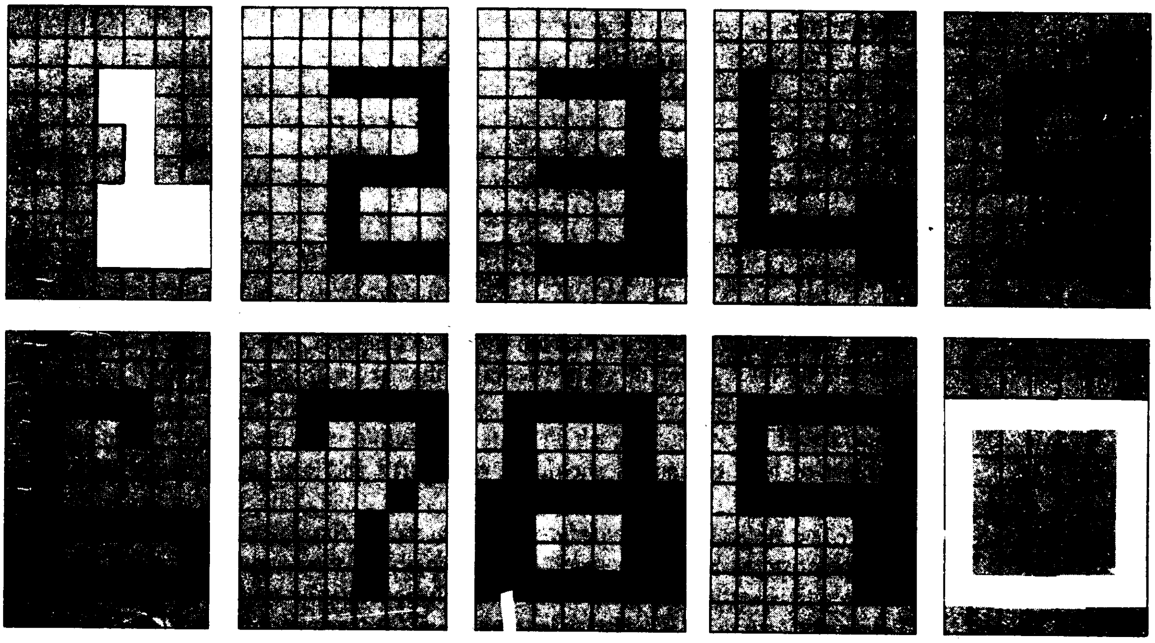
9.34 The control of a single-address small computer normally passes through two major phases in executing an instruction which fetches a single operand from memory (an ADD or SUBTRACT instruction, for example). We call these the *instruction cycle* and the *execution cycle*. In order for control to know which phase or cycle it is in, a conventional random logic control unit uses an E flip-flop and an I flip-flop.

(*a*) Why are two flip-flops used instead of one?

(*b*) Why does a microprogrammed version of the same computer not require an E and an I flip-flop?

9.35 Write the transfer in Table 9.3 in register transfer language. Write the transfers for a SUBTRACT instruction (as in Table 9.2) in register transfer language.

9.36 Write a register transfer statement to transfer every other bit (starting with bit X_0) from a register X with 10 flip-flops into a register Y with 5 flip-flops.



COMPUTER ORGANIZATION

Computers are available in a wide range of sizes and capabilities. The smallest computers are called microcomputers, the next largest are minicomputers, followed by small, medium-sized, and finally the large, super, or "maxi" computers. The prices range from a few dollars (for a chip set for a microcomputer) to several million dollars. Speeds are from microseconds per instruction to hundreds of instructions per microsecond.

A microcomputer generally consists of several integrated-circuit (IC) chips, including a central processing unit (CPU) chip (or chips), called a *microprocessor chip* (or chips); several memory chips; and one or more input-output interface chips. These sets of chips can be quite inexpensive (a few dollars in large quantities) or fairly expensive (several hundred dollars for high-speed chip sets). Hand calculators are often assembled from IC chips including one of the lower-priced microprocessor chips. Personal computers also use microcomputer chip sets.

Microprocessor chips also are widely used in so-called original equipment manufacturer (OEM) devices or systems. Traffic lights, printers, communications controllers, automatically controlled instrument complexes, cash registers, and automatic checkout facilities in grocery and department stores, for example, all make wide use of microprocessors.

Similarly, minicomputers, which generally have prices from a thousand to tens of thousands of dollars (including memory and input-output devices), are widely used in control systems and OEM systems as well as in scientific applications and business data processing for small businesses, schools, laboratories, etc. The minicomputer preceded the microcomputer, and it continues to be widely used since computers in this price range provide many users with enough additional capabilities to warrant the extra cost.

The small- and medium-scale computer market finds applications in businesses and laboratories of all kinds as well as in hospitals, warehouses, small banks, etc. The largest computers are to be found in large corporations such as insurance companies, banks, scientific laboratories, and universities. These "super" computers range from scientific application oriented "number crunchers" to large complexes of input-output devices and memories used in businesses where emphasis is on maintaining large files of data, producing management reports, billing, automatic ordering, inventory control, etc.

The characteristics of these different kinds of computers differ considerably from category to category and from design to design. Computers for business data processing have different system features than those for scientific work. There is also considerable variation in opinion as to how computers for the same application area should be configured, which leads to differing computer designs. The general subject of how computers should be configured and what features should be included is called *computer architecture.*

The subject of *computer architecture* ranges through almost every aspect of computer organization. Included are the lengths of the instruction words, whether the length is variable or there are several different lengths, and how many addresses in memory are referenced by an instruction word. Other architectural considerations concern the number of bits in each memory word, whether instructions and data words are of the same size as the memory words, whether numbers are handled in 1s or 2s complement form or in BCD or some combination of these. What are the instructions provided, how are the memories organized, and how are input-output devices interfaced? As can be seen, computer architecture is a large and rich subject which deals with most aspects of computer design and organization and interacts with every aspect of the computer.

OBJECTIVES

1 The addressing techniques used in computers are explained, and examples from existing computers used to illustrate these techniques.

2 Good programming practice calls for breaking programs into subprograms. A single subprogram can be used several times in an overall program. Computer instruction repertoires have special instructions to go to and return from subprograms, and these are described along with how they work.

3 Interrupts from I/O devices are handled differently by various computers. The general principles involved are discussed, and examples from computers presented.

4 The architectures and instruction repertoires for several present-day computers are described. Examples of sections of programs for these computers and explanations for their operation are included.

INSTRUCTION WORD FORMATS—NUMBER OF ADDRESSES

10.1 A given computer has one or more basic formats for its instruction words. We have emphasized the single-address instruction word, which is popular for microcomputers. There are also several other formats in use.

Two-address instructions

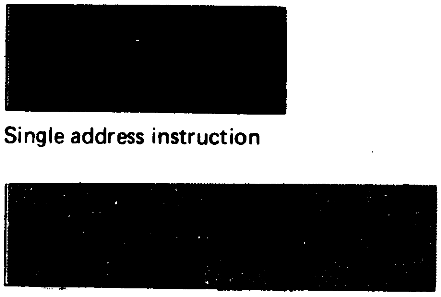
The number of divisions in the basic computer instruction word is determined primarily by the number of addresses which are referred to. The single-address instruction has been covered. Many computers, however, have two-address instruction words with three sections (Fig. 10.1), the first consisting of the OP code and the second and third sections each containing the address of a location in the memory.

Different computers use these addresses differently. Generally, both addresses in a two-address machine specify operands, and the result is stored at the first address.[1] The Minneapolis-Honeywell 200 and the IBM 370 and 1801 series have two-address instructions that provide examples in which each address refers to an operand in the memory.

In many computers, instead of a single accumulator, there are two or more registers which are called either *multiple accumulators* or *general-purpose registers*. An instruction word will have the first address section (the "address of operand A" section in Fig. 10.1) tell which general register contains one of the operands. The second address section of the instruction word will then give the address in memory of the second operand. If only two accumulators or general registers are provided, only 1 bit is needed for the address section; if 16 general registers are used, then 4 bits will be needed for the first address. Results are generally stored in the general register (accumulator) specified by the first address.

In some computers instruction words are provided in which each of the two addresses refers to general registers. Thus, for instance, in a computer with 16 general registers, an instruction word would consist of the OP code plus two 4-bit address sections.

[1] In several computers the result of the calculation is stored at the second of the two addresses.



Single address instruction

Two address instruction

FIGURE 10.1

Formats for instructions.

Two-address instruction words in which one or both addresses refer to general registers are shorter than two-address instruction words where both addresses refer to the memory, and this format is popular in microcomputers and minicomputers.

Zero-address instructions—stacks

There is a type of instruction word that does not specify any location in memory for an operand, but which relies on what is called a *stack* to provide operands. Basically a stack is a set of consecutive locations in a memory into which operands can be placed. The name *stack* is derived from the fact that the memory is organized like a stack of plates in a cafeteria. (Each operand can be thought of as a plate.) The first operand placed on the stack is said to be at the *bottom* of the stack. Placing an operand on the stack is called *pushing*, and removing an operand is called *popping* the operand. The operand most recently placed on the stack is said to be on the *top* of the stack. Only this top operand is immediately available.

If we push operands *A*, *B*, and *C* onto an empty stack and then pop an operand, *C* will be removed. If we push *A*, *B*, and *C* in order and then pop three operands, first *C* will be popped, then *B*, and finally *A*. (This last-in first-out principle leads to stacks sometimes being called *LIFO lists*.)

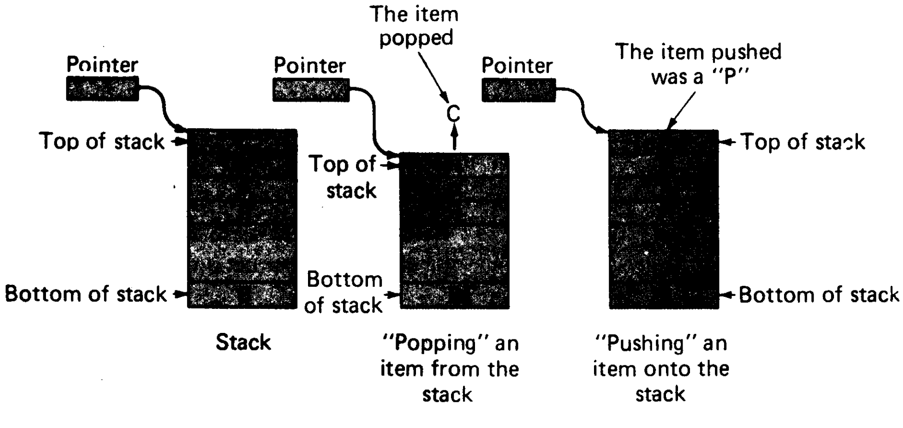
Figure 10.2 shows the operation of a stack. Stacks are generally maintained as a set of words in a memory. Each word therefore has a fixed length (number of bits) and an address. The *stack pointer* is a register that contains the address of the top operand in the stack. The stack pointer is incremented or decremented when an operand is pushed or popped.

If an ADD instruction is given to a computer using a stack architecture, the top two operands in the stack will be removed; added and then the sum is placed on the top of the stack. Similarly, a MULTIPLY instruction would cause the top two operands to be multiplied and the product placed on the stack.

Since only the OP-code section of an arithmetic instruction need be given to specify an arithmetic operation, these instruction words can be very short. It is still

FIGURE 10.2

Stack operations.



Note: In this case each item in the stack is a single character. The stack items could be numbers, words, records, etc. The pointer contains the address of the "top of the stack."

necessary, however, to move operands from memory onto the stack and from the stack back into the memory, and the instruction words for this will be longer since memory addresses must be specified. (These instruction words will be like single-address instruction words, except that the operands are moved to and from the stack instead of to and from the accumulator.)

The advocates of stacked computer architecture have some convincing arguments, but problems do exist. Stacked computers include the Burroughs 5500 and 1700 and the Hewlett-Packard 3000. Stacks are widely used in other sections of computers we show.

REPRESENTATION OF INSTRUCTIONS AND DATA

10.2 Important features of a computer's architecture concern the number of bits in instruction words, the size of memory words, and the way data are represented in the computer. In most early computers and in some present-day computers, the high-speed memory contains the same number of bits at each address (in each location) as the instruction words. Similarly, numbers are represented by using the same number of bits. This makes for straightforward implementation. An example of a computer with this structure can be found in the 6100, which has a 12-bit/word memory, 12-bit instruction words with a 3-bit OP code, and numbers represented by using a 12-bit signed 2s complement number system. Most of the large scientific number-crunching machines also use this structure, and CDC produces a number of 64-bit/word large computers with this basic structure as well as some smaller 24-bit/word computers. CRAY also makes computers of this type.

There is a desire to be efficient with the length of instruction words. Also, business data processing involves much manipulation involving character strings (names, addresses, text, etc.). The desire to create computer architectures that conserve on instruction word length and also permit storing of strings of characters of arbitrary length efficiently has led to a number of computer architectures with (1) only 8 bits at each address in memory, so a single alphanumeric character can be stored at each address, and (2) instruction words with variable lengths (each word length is some multiple of 8 bits).

As a result, most small computers now have memory words of 8 bits per word. Instruction words are then of variable length with each being some multiple of 8 bits. Data words are also multiples of 8 bits with many microprocessors having 8-, 16-, and 32-bit words.

ADDRESSING TECHNIQUES

10.3 When an address in memory is given in an instruction word, the most obvious technique is simply to give the address in binary form. This is called *direct addressing*, and the instruction words in the examples in Chap. 9 all use direct addressing.

Although direct addressing provides the most straightforward (and fastest) way to give a memory address, several other techniques are also used. These techniques are generally motivated by one of the following considerations:

1 *Desire to shorten address section* For instance, if we have a computer with a 256K memory, 18 bits will be required for each direct address, some addressing techniques are used to reduce this number.

2 *Programmer convenience* Several addressing techniques (such as index registers, which are described) provide a convenience to the programmer in writing programs.

3 *System operation facilities* In many computer systems, the computer will have several different programs in memory at a given time and will alternate the running of these programs. To efficiently load and remove these programs from memory in differing locations, addressing techniques are provided which make the program *relocatable,* meaning that the same program can be run in many different sections of memory. The operating systems in microcomputers use this facility.

The following sections describe the basic addressing techniques now in use: direct addressing, immediate addressing, paging, relative addressing, indirect addressing, and indexed addressing. Examples of these techniques are given for actual computers and enough are given so the principle can be clearly understood. Knowing how real computers use these techniques is important.

DIRECT ADDRESSING

10.4 Simply giving the complete binary address in memory is the most direct way to locate an operand or to give an addrèss to jump to. As a result, most computers have some form of *direct addressing*. The following examples are for computers that will also be used in the sections on more complex addressing strategies.

Example

The 8080 microprocessor has a single 8-bit accumulator. The 8080's memory is organized into words of 8 bits each which are called *bytes*. An OP code for this microprocessor occupies 8 bits, or 1 byte, an entire memory location. The address bits are then located in the following memory locations. Since 2^{16} words can be used in a memory, 2 bytes are required for a direct address. As a result, a direct-address instruction requires 3 bytes in memory—one for the OP code and two for the direct address.

In executing an instruction, the 8080 CPU[2] always obtains the OP code from memory first, and this tells how many bytes are required for the address. The 8080 CPU then reads the necessary bytes from memory and assembles a complete instruction word in its registers, which it then proceeds to execute.

A typical direct-access instruction in the 8080 is the LDA (load accumulator) instruction with OP code 00111010 (3A hexadecimal). This OP code is followed

[2] The 8080 CPU is constructed on a single chip. This chip, which is sometimes called the *8080 microprocessor chip,* interprets and executes instructions. Memory is on separate chips, as are input-output interface circuits.

by 2 bytes giving the address in memory of the 8-bit word to be loaded into the accumulator. The low-order (least significant) bits of the address are given in the first byte of the address and the high-order bits in the second byte.

Assume that the memory contains these values:

ADDRESS (HEXADECIMAL)	CONTENTS (HEXADECIMAL)
0245	3A
0246	49
0247	03
...	...
0349	23

The 3 bytes in locations 245, 246, and 247 contain a single LDA instruction which, when executed, will cause the value 23_{16} to be transferred into the accumulator of the 8080 microprocessor.

Example

The 6800 microprocessor has two 8-bit accumulators which are referred to as accumulator *A* and accumulator *B*. The microprocessor has 8 bits per memory word. The OP code of an instruction occupies 8 bits and therefore a complete memory word. The address bits for an instruction word are in the memory location(s) following the OP code. The memory can be up to 2^{16} locations in size. As an example of direct addressing, the OP code for ADDA, which causes the contents of the address referenced to be added to and then stored in accumulator *A*, is BB (hexadecimal), or 10111011 (binary). If the microprocessor reads this OP code, it knows that the address is given in the following 16 bits. As a result, if the 6800 CPU reads an OP code of BB, it then reads the next 2 bytes in memory to obtain the address. The microprocessor reads from this address and performs the required addition. The next OP code is read from the memory location following the two locations that contained the address.[3]

The OP code for an ADDB instruction, which causes the number stored in the memory location referenced by the next 16 bits to be added to accumulator *B*, is FB (hexadecimal). Now examine Fig. 10.3. If the microprocessor reads the two instruction words shown, it will cause addition into first accumulator *A*, then accumulator *B*, and will take the next instruction word from location 17 in the memory.

In the 6800 microprocessor, instruction words can have addresses with 1 byte or 2 bytes, as will be seen. [Some instructions have only "implied addresses" (no address bits); HALT is such an instruction.] The microprocessor must therefore read the OP code before it can determine how many more locations from the memory need to be read to form the instruction word.

One note is necessary here. The 6800 microprocessor also has instructions with only 8-bit addresses. In this case an address has only 8 bits, and thus only

[3] Notice that the 6800 places the most significant bits in the address in the second byte and the least significant bits in the third byte. (The 8080 does the reverse.)

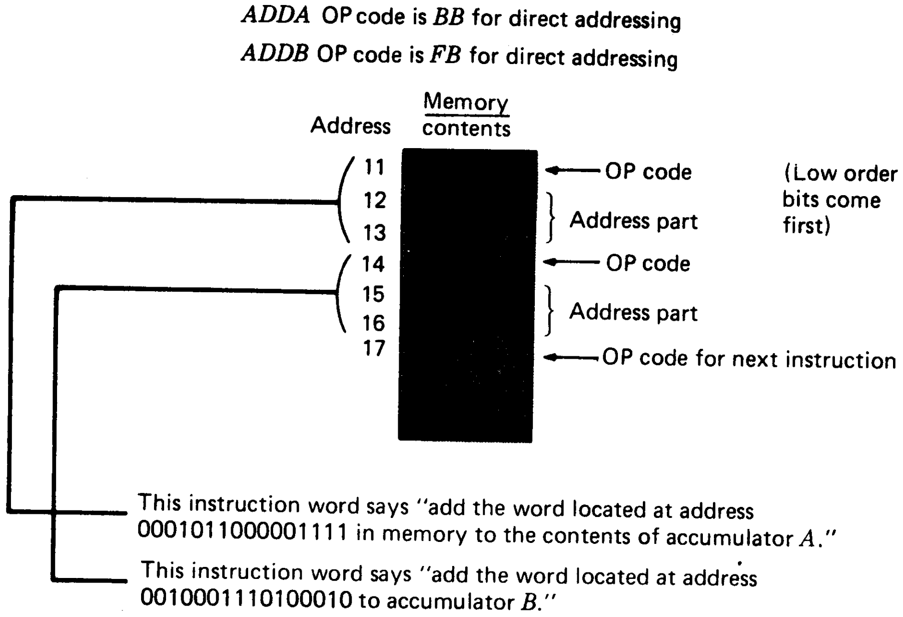

ADDA OP code is *BB* for direct addressing
ADDB OP code is *FB* for direct addressing
Memory contents
Address
11
12
13
14
15
16
17
OP code
Address part
OP code
Address part
OP code for next instruction
(Low order bits come first)
This instruction word says "add the word located at address 0001011000001111 in memory to the contents of accumulator *A*."
This instruction word says "add the word located at address 0010001110100010 to accumulator *B*."

FIGURE 10.3

6800 microprocessor instruction execution.

the first 256 bytes in the memory can be referenced. As an example, an instruction to add the number at the location given in the following 8 bits to accumulator A has OP code 9B. An instruction to add to accumulator B the number at the location given in the following byte has OP code DB. The OP code tells whether a complete 16-bit address or an 8-bit address is to be read from the memory. (In its manuals Motorola calls the 8-bit address instruction words *direct-addressing instructions* and the 16-bit address instruction words *extended direct-addressing instructions.*) The 16-bit addresses have been used to illustrate the direct-addressing technique because they are more natural and all the memory can be reached.

Example

The PDP-11 is a DEC minicomputer and microcomputer series with sizes ranging from small to large. A particular size is designated by the model number, so that a PDP-11/05 is a small computer, the PDP-11/45 is a medium-sized machine, and the PDP-11/70 is a fairly large system. This series of computers is typical of what is offered by DEC and other manufacturers.

The PDP-11 has eight 16-bit general registers (accumulators). It is common practice to name these general registers R_0 to R_7, and we follow this practice.

The PDP-11 memory is organized into 8-bit words, so 1 byte is in each memory location. The PDP-11 has a number of addressing modes and, as a result, a fairly complex instruction word format.

A typical direct-address instruction in the PDP-11 involves adding the numbers in two general registers and storing the sum in one of the registers. The instruction word to do this has three sections: the OP code, the source address, and the destination address. (In an ADD, the number in the source register is added to that in the destination register, and the sum is placed in the destination register.) Since the source and destination are each general registers and there are eight general registers, 3 bits are required to give each address. However, since the PDP-11 has a number of addressing modes, 3 extra bits are included in each of the source and destination addresses to tell which addressing mode is to be used. The instruction word format is as follows:

OP code	Source address	Destination address
15 12	11 6	5 0

The first (leftmost) 3 bits in the source and destination addresses give the mode, and for direct addressing these will be all 0s. The next 3 bits give the register number. The OP code for ADD in the PDP-11 is 0110, and so the instruction word which will add register 3 to register 5 and store the sum in register 5 is

0110	000011	000101

Another example of direct addressing is the increment instruction, which simply adds 1 to a selected general register. The instruction word to accomplish this has two sections: an OP code and an address section. The address section has 3 bits to tell the mode and 3 bits to designate the register. The OP code for an INC (increment) instruction is 0000101010. Thus an instruction that will increment general register 5 is

0000101010	000101
OP code	address part

Notice that this OP code is larger than the ADD OP code, because only one operand is required here. (The first 4 bits are not duplicated in any of these larger OP codes; they tell the class of operation.)

IMMEDIATE ADDRESSING

10.5 A straightforward way to obtain an operand is simply to have it follow the instruction word in memory. Suppose that we want to add the number 7 to the accumulator in a single accumulator computer, and suppose that the memory is organized in 8-bit bytes. A direct way to cause this addition would be to have an 8-bit OP code which says to ADD and that the augend follows "immediately" in memory (the next byte). The computer would then read the OP code, get the byte to be added from memory (which would contain 7), add it into the accumulator, and take the next instruction word's OP code from the byte following the byte containing the augend. This is essentially how the 8080 (and 6800) computers operate.

In general, immediate addressing simply means that an operand immediately follows the instruction word in memory.

Example

For the 8080 microprocessor, the instruction ADI (add immediate) has OP code 11000110 and tells the CPU to take the byte following this OP code and add it into the accumulator. Consider the following:

ADDRESS	CONTENTS
16_{16}	11000110
17_{16}	00001100
. . .	. . .

When the computer reaches address 16_{16} in memory, it reads the OP code, sees that this instruction is an ADI instruction, takes the next byte from the memory which is 00001100, adds this into the accumulator, and takes the next OP code from location 18_{16} in memory.

Example

The 6800 microcomputer has two accumulators, and so the OP code must tell which accumulator to use. The instruction ANDA with OP code 84_{16} will cause the byte following the OP code to be ANDed bit by bit with accumulator A, while the instruction ANDB with OP code $C4_{16}$ will cause the byte following the OP code to be ANDed bit by bit with accumulator B.

Suppose that accumulator A contains 01100111 and accumulator B 10011101. Then consider this in memory:

ADDRESS	CONTENTS
10_{16}	10000100
11_{16}	11010101
12_{16}	11000100
13_{16}	10100101
14_{16}	. . .

The 6800 will read the ANDA at location 10_{16} and AND the next byte with accumulator A, giving 01000101, which will be placed in accumulator A. It will then read the ANDB in location 12_{16}, AND the next byte with B to give 10000101, place this in accumulator B, and read the next OP code for an instruction from location 14_{16}.

Example

The PDP-11 has eight accumulators and so must tell which accumulator to use when an addition instruction uses immediate addressing. The OP code for ADD is 0110, and an instruction word for an immediate add looks like this:

0110	010111	000011
OP code	source	destination

The source bits say that the add is an immediate add and that the augend is in the 2 bytes (since the accumulators have 16 bits) following this word. The destination section here refers to general register 3, so the next 16 bits will be added into general register 3. (Placing 101 in the rightmost bits instead of 011 will cause an addition into general register 5, etc.)

Now consider that general register 4 contains 000061_8, and the memory is as follows (all these numbers are in octal, which is DEC's practice):

ADDRESS	CONTENTS
1020–1021	062704
1022–1023	000012
...	...

Execution of these by the CPU will result in 12_8 being added into general register 4, giving 73_8 in that register.

Notice in the PDP-11 that the ADD instruction OP code is the same for immediate and for direct addressing. It is the first 3 bits in the source and destination address sections that tell the addressing mode, not the OP code.

PAGING

10.6 Microcomputers and minicomputers sometimes alleviate the problem of addressing a large memory with a short word by using a technique that actually arose in a large computer called Atlas. This technique is called *paging*. When paging is used, the memory is divided into pages, each of a fixed length. An instruction word then designates a page and a location on that page.

Example

For the 6100 microprocessor mentioned in Chap. 9, the basic memory of 4096 words of 12 bits each is divided into 32 *pages* of 128 words each. Thus page 0 contains the memory locations from 0 to 127 (decimal), page 1 refers to the memory locations from 128 to 255, . . . , and finally page 31 refers to the locations from 3968 to 4095, as shown in Fig. 10.4. Then 5 bits are required to reference a page, and 7 bits to reference a location within a page.

The addressing of data in a computer with paging varies from computer to computer. Generally the address given in the instruction word can refer either to the page in which the instruction word lies or possibly to some other particular page previously specified.

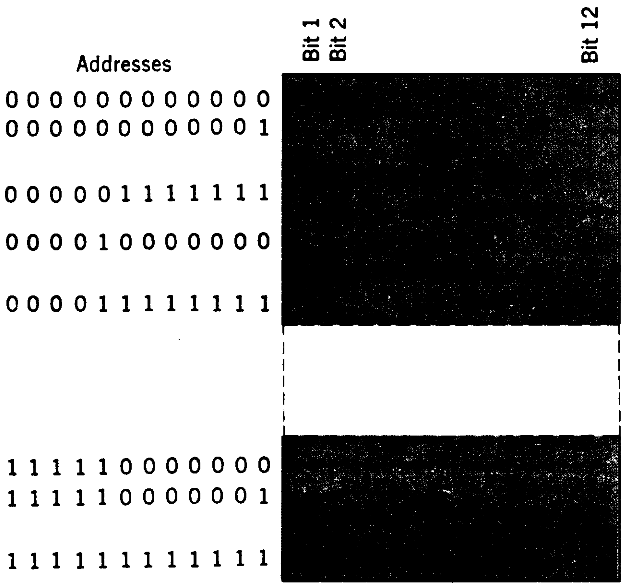


FIGURE 10.4

Layout of 32-page memory with 4096 words of 12 bits.

Example

For the 6100 the seventh bit in a 12-bit instruction word is called the *page bit.* This bit tells whether the 7-bit address in the instruction word refers to the page in which the instruction lies (in which case the bit is a 1) or to the first page in the memory (in which case the bit is a 0).

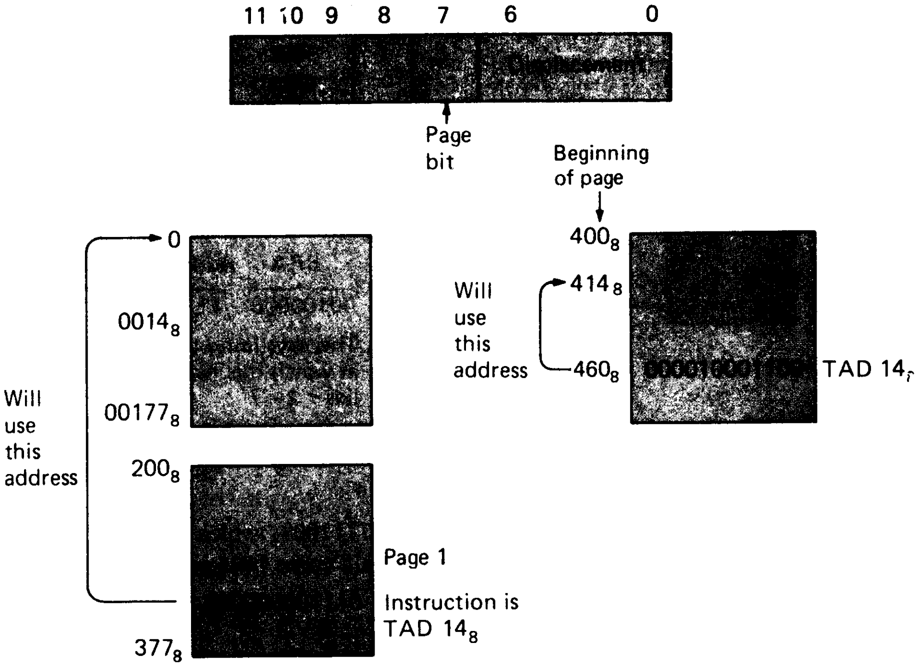
Figure 10.5 shows the page bit in the instruction word for the 6100. If this bit is a 0 in a TAD (2s complement ADD) (or other) instruction, the address given by bits 6 to 0 refers to an address on page 0 (see Fig. 10.4). Therefore the instruction word 001000001110 refers to location 14_8 in memory, regardless of where in memory the instruction is placed. (The first 3 bits are 001, the OP code for TAD.)

If the instruction word has a 1 in the page bit position and if the final bits give the number 14_8, then the address to be used is the 15_8th address on the page in which the instruction lies. This is shown in Fig. 10.5 where the instruction word 001010001100, when located at address 000100110000 in memory, points to location 000100001100 in memory; and the augend for the TAD would be fetched from that address.

Paging shortens the length of the address part. For the 6100, a 4096-word memory, which would require 12 address bits if a direct address were used, is addressed by using a page bit and 8 more bits. Some addresses are not reachable from a given instruction word, however. To reach addresses not on page 0 or the page containing the instruction word, a technique called *indirect addressing* is used, as will be shown.

RELATIVE ADDRESSING

10.7 *Relative addressing* is quite similar to paging, except that the address referred to is relative to the instruction word. In general, when relative addressing is used, the address part of the instruction word gives a number to be added to the



RELATIVE ADDRESSING

FIGURE 10.5

Paging examples.

address following the instruction word. Thus in relative addressing, the address section contains a displacement from the instruction word's location in the memory. Giving only a displacement reduces the number of address bits but makes only a part of the memory available. For instance, if the instruction word uses relative addressing and the address part contains 8 bits, then only 256 memory locations are available to a given instruction.

Relative addressing is best explained by using examples.

Example

The 6800 microprocessor can have up to 2^{16} memory words. So 16 bits are required to address the entire memory in a direct mode. When relative addressing is used, this address is reduced to an 8-bit displacement, shortening the instruction word.

In the 6800 a relative-address instruction word contains only the OP code and an 8-bit address, so only two locations in memory (bytes) are required. (The OP code tells what kind of addressing to use.) When the addressing is relative, the address in the second byte of the instruction is added to the address at which the OP code lies plus 2. The address in the second byte is considered as a signed 2s complement number, however, so the address referenced can be at a higher or lower address in memory than the instruction word. In fact, the address can be from -125 to $+129$ memory words from the address of the OP code.

Figure 10.6 shows this. The OP code for a BRA (branch) instruction with relative addressing is 20 (hexadecimal). The microprocessor would read the OP code at location 10, see that it is a BRA instruction, get 00000101 from the next memory location, add this 5 (decimal) to 2 plus 10 (where the OP code lies), giving 17. The next OP code would then come from location 17. In Fig. 10.6 this OP

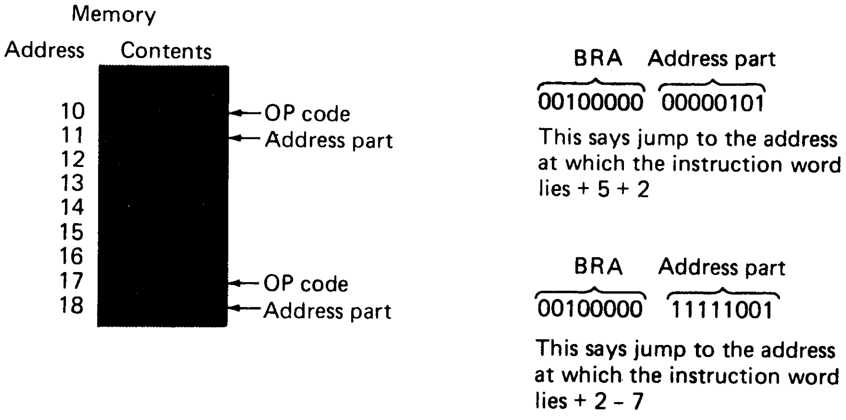

Memory
Address
Contents
10
11
12
13
14
15
16
17
18
OP code
Address part
OP code
Address part
BRA
Address part
00100000
00000101
This says jump to the address at which the instruction word lies + 5 + 2
BRA
Address part
00100000
11111001
This says jump to the address at which the instruction word lies + 2 - 7

FIGURE 10.6

Relative addressing in 6800 microprocessor.

code is again BRA, and location 18 contains 11111001, which is −7 decimal. Since 17 + 2 − 7 = 12, the next OP code would come from location 12.

Example

In the PDP-11, a relative addressing mode can be used for the INC (increment) instruction. The OP code for INC in the PDP-11 is 0052_8, and 27_8 in the address part indicates a relative addressing mode.

Assume that we have the following situation in memory:

ADDRESS (OCTAL)	CONTENTS (OCTAL)
1020	005627
1022	000012
1024	. . .

The relative addressing feature operates as follows. The displacement, 12_8 in this example, is added to the address following the instruction word, in this case 1024. This gives 1036, and so the number at location 1036 in memory would be incremented.

INDIRECT ADDRESSING

10.8 Another widely used variation in addressing is called *indirect addressing*. Indirect addressing causes the instruction word to give the address, not of the operand to be used, but of the address of the operand. For example, if we write ADD 302 and the instruction is a conventional direct-addressing addition instruction, the number at location 302 will be added-to the word currently in the accumulator.

If the addition instruction is indirectly addressed and we write IAD 302 (indirect add), then the number stored at address 302 will give the *address* of the operand to be used. As an example, when the instruction word at address 5 in the memory in the following program is performed, it will cause the number 164 to be added to the current contents of the accumulator.

MEMORY ADDRESS	CONTENTS
5	IAD 302
. . .	. . .
302	495
. . .	. . .
495	164

INDIRECT ADDRESSING

Example

The 6100 has a 12-bit instruction word, as shown in Fig. 10.7. The instruction word contains a 3-bit OP code, a page bit, an indirect bit, and a 7-bit unsigned binary number. The page bit has been explained. There is also an indirect bit. A 0 in the indirect bit says, "This is not an indirect address." However, a 1 in the indirect bit indicates the address referenced is indirect.

As an example refer to Fig. 10.7. The TAD instruction with OP code 001 is used again. The figure shows a case in which the instruction word is TAD with the page bit a 0, so the address is on page 0 in the memory, and the indirect bit is a 1. Since the 7 address bits point to location 62_8, that location contains the address of the operand. Again, referring to the figure, at location 62_8 we find a 3432_8, and it is this location, at which the operand 3255, is located, that is added to the accumulator.

Notice that the entire 4096-word memory is accessible to an instruction word through the use of indirect addressing because each memory word contains 12 bits. Placing an address on page 0 where it can be reached from anywhere in the memory makes that address always available in programming.

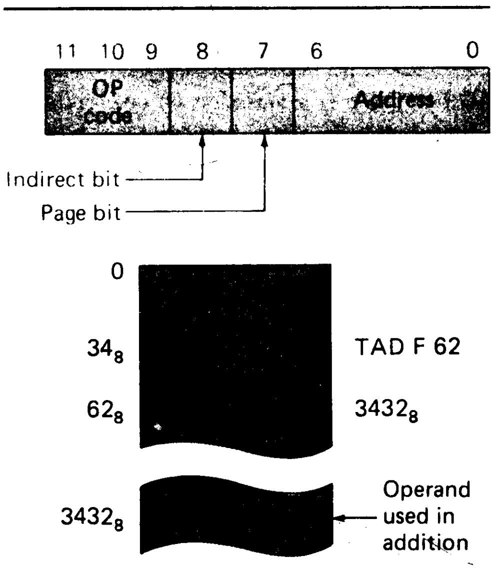


FIGURE 10.7

Indirect addressing in 6100 microprocessor.

Example

In the 8080 microprocessor there are several registers in the CPU in addition to the accumulator. These are called *scratchpad registers* and may be used in several types of instructions. The scratchpad registers are named B, C, D, E, H, and L. The registers are each 8 bits in length. Sometimes they can be handled in pairs with a resultant length of 16 bits, thus forming a complete address. Then an indirect-address mode can be used where the number in the register pair points to the address where the operand lies.

For example, in the 8080 there is a MOV (move) instruction which moves an 8-bit word from the memory into a designated register. The format for this instruction word is as follows:

$$\overbrace{0\ 1}^{\text{OP}}\ \overbrace{D\ D\ D}^{\text{destination}}\ \overbrace{1\ 1\ 0}^{\text{source}}$$

The DDD section here is 3 bits,[4] which simply call out the register into which the 8-bit word from memory is to be moved. The accumulator has number 111, register B has number 000, scratchpad register C has number 001, etc. The location in the memory from which the word to be moved is taken is always given by the register pair H, L. Thus if register pair H, L contains $45A2_{16}$ (H contains 45, L contains A2), then the address in memory used will be $45A2_{16}$.

If the instruction word is 01111110 and register pair H, L contains 3742_{16}, then the word at location 3742_{16} will be moved into the accumulator. If the instruction word is 01001110 and the H, L pair contains 2379_{16}, then the word at location 2379_{16} in memory will be moved into scratchpad register C.

Moving an address into the H, L registers in the CPU makes the word at that address available to an instruction word through the use of indirect addressing. Complete details and how to load the H, L pair are given in Sec. 10.11.

INDEXED ADDRESSING

10.9 There is a variation on conventional direct memory addressing which facilitates programming, particularly the programming of sequences of instructions that are to be repeated many times on sets of data distributed throughout the machine. This technique is called *indexing*.

Indexing was first used in a computer developed at the University of Manchester. A register named the *B box* was added to the control section.[5] The contents of the B box could be added to the contents of the memory address register when desired. When the B box was used, the address of the operand located in memory would be at the address written by the programmer plus the contents of the B box. The U.S. term for B box is *index register*, and we use this term. Index registers are so useful that computers sometimes provide several.

[4] Each D stands for a single destination bit that can be either a 0 or a 1.

[5] The idea was so useful that several B boxes were later used.

INDEXED ADDRESSING

Use of index registers eases writing programs that process data in tables, such as those described in Chap. 1, greatly reducing the number of instructions required in an iterative program. The index registers permit the automatic modification of the addresses referred to without altering the instructions stored in memory.

When index registers are included in a computer, a section of the instruction word tells the computer whether an index register is to be used and, if so, which one. So the basic instruction word is broken, for a single-address computer, into three parts instead of two. A typical division is shown in Fig. 10.8.

Generally two additional instructions are also added. One is used to load the index register, and the other modifies the number stored in the specified index register or causes the computer to branch.

If an index register is not to be used, the programmer places 0s in the index register designation section of the word. If there are three index registers, there will be two binary digits in the index register section of the word, and the index register desired can be selected by placing the correct digits in the index register designation section.

To describe the operation of the index registers, two instructions are introduced. We designate one of these by the mnemonic code SIR (set index registers), and this instruction will cause the address section of the instruction word to be transferred into the index register designated by the index register designation bits in the word. For instance, 01 SIR 300 will load the number 300 into index register 01. Since the address section normally contains the address section of the computer word, all that is required is that the contents of the address register be transferred into the index register designated.

We designate the second instruction with the mnemonic code BRI (branch on index). This instruction will cause the contents of the index register designated to be decreased by 1 if the number stored in the index register is positive. At the same time, the computer will branch to the address in the address section of the instruction word, taking its next instruction from that address. If the index register designated contains a 0, the computer will not branch but instead will perform the next instruction in normal order.

The index registers may be used during any normal instruction by simply placing the digits indicating the index register to be used in the index register designation section of the computer word. For instance, if index register 01 contains 300 and we write a CAD (clear and add) instruction

01 CAD 200

then the computer will add the contents of index register 01 to the contents of the address section, and the address used will be the total of these two. Since index

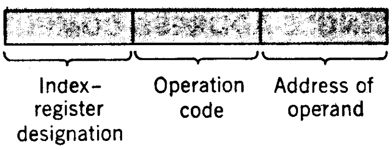


FIGURE 10.8

Index register instruction word.

TABLE 10.1

ADDRESS IN MEMORY	INSTRUCTION WORD			COMMENTS
	INDEX REGISTER DESIGNATION	OP CODE	ADDRESS SECTION	
0	01	SIR	99	The number 99 is placed in index register 01.
1	01	CAD	201	Picks up number to be added.
2	00	ADD	301	Adds to total thus far.
3	00	STO	301	Stores the current sum.
4	01	BRI	1	Subtracts 1 from index register 01 and then branches to first instruction until index register 01 contains 0, then proceeds to next instruction.
5	00	HLT	0	
201 to 300	Contain numbers to be added.			
301	Location at which sum is stored.			

register 01 contains 300 and the address section contains 200, the address from which the operand will be taken will be address 500 in memory.

An example of the use of an index register may be found in the short program shown in Table 10.1, which will add all the numbers stored in memory addresses 201 to 300 and store the sum in address 301.

The program repeats the instructions at addresses 1 to 4 until index register 01 is finally at 0. Then the computer does not branch and is halted by the next instruction.

Example

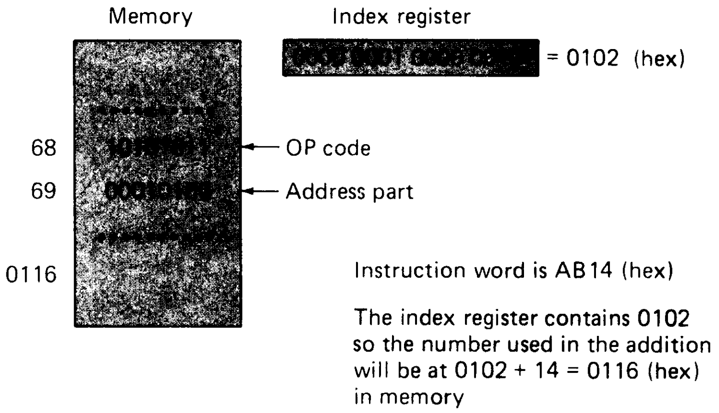
The 6800 microprocessor mentioned before has a single 16-bit index register. For the ADDA instruction, when indexing is used, the OP code is AB (hexadecimal). This instruction has only one 8-bit address part, so an entire instruction word requires only 16 bits (two memory locations).

Figure 10.9 shows an example for the 6800 microprocessor. The instruction word is at locations 68 and 69 in memory and is an indexed ADDA instruction. The 8-bit address part contains 14 (hexadecimal), and the index register in the CPU contains 0102. This results in the number at location 116 in memory being added into accumulator *A*.

The 6800 has instructions to load, increment, or decrement the index register, and these are covered in Sec. 10.12.

SINGLE-ADDRESS COMPUTER ORGANIZATION

10.10 A straightforward example of a single-address computer is the Harris 6100, which is widely used in word processors and also personal computers and I/O device interfaces. This computer is used to point out some basic principles in computer architecture.



Instruction word is AB14 (hex)

The index register contains 0102 so the number used in the addition will be at 0102 + 14 = 0116 (hex) in memory

FIGURE 10.9

Index register in the 6800 microprocessor.

The basic organization of the 6100 is shown in Fig. 10.10. It has a 12-bit instruction word as shown in Fig. 10.11. The memory is also organized in 12-bit words, and a basic memory block consists of $2^{12} = 4096$ words. Since the instruction word is very short, the designers have allowed only 3 bits for the OP-code part; thus, as shown in Fig. 10.11, there are only eight basic classes of instruction words.

Several of the basic instructions are very straightforward. For instance, the TAD instruction (for 2s complement ADD) simply performs a 2s complement addition of the operand addressed in memory to the operand currently in the 12-bit single accumulator and places the sum in this accumulator. There is an extra flip-flop called the *link*, or *L, bit* which receives any overflow from this addition. The OP code for TAD is 001_2, or 1_8.

Similarly, the AND instruction simply performs a bit-by-bit AND on the operand addressed in memory and the contents of the accumulator, placing the result in the accumulator. The OP code for AND is 000_2, or 0_8.

The addressing techniques used in the 9 bits to form addresses have been described. The eight bit in an instruction word is a 0 if the address is direct and a 1 if indirect (the address of the address). The seven bit is a 0 if the remaining 7 bits (bits 0 to 6) give the actual address on page 0 of the memory and a 1 if the address is on the same page as the instruction word.

An ISZ (increment and skip if zero) instruction simply increments the word addressed and returns it to memory. However, if this word becomes a 0, the next instruction following the ISZ instruction is not executed. Instead, the computer next executes the word in memory following this word. This particular instruction is very useful in indexing through tables when indirect addressing is used and is provided as a substitute for index registers.

The DCA (deposit and clear accumulator) instruction stores the accumulator in the memory at the address specified and also clears the accumulator to all 0s. The OP code for DCA is 011_2, or 3_8.

The JMS (jump to subroutine) instruction brings up an important feature in instruction repertoires for computers. It is good form in writing a computer program to break the program into as many subprograms (separable pieces) as possible.

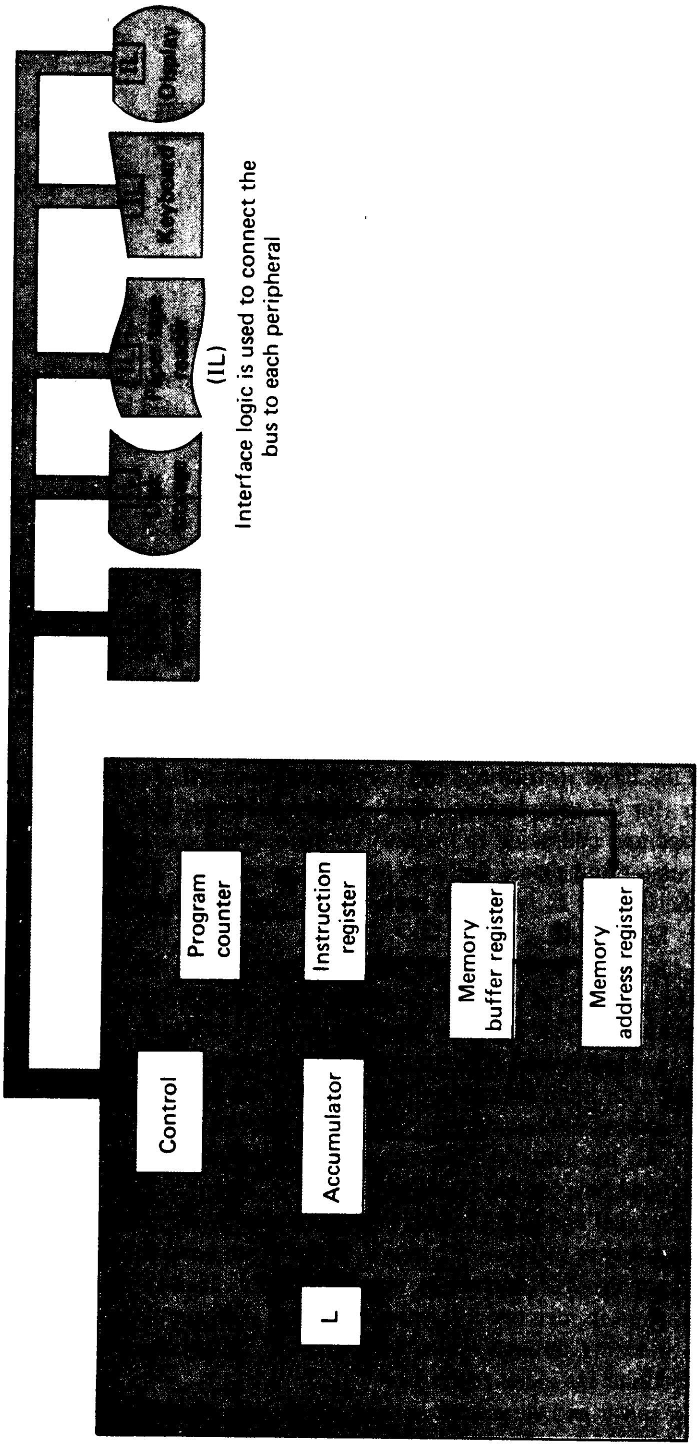
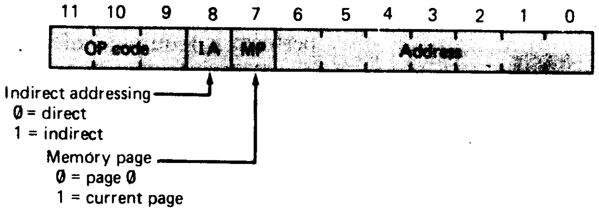


FIGURE 10.10

Organization of the 6100 microprocessor.

BASIC INSTRUCTIONS

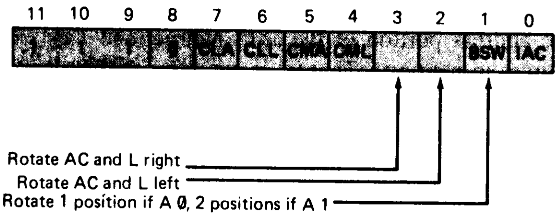
AND	0000	Logical AND
TAD	1000	2s complement add
ISZ	2000	Increment, and skip if zero
DCA	3000	Deposit and clear AC
JMS	4000	Jump to subroutine
JMP	5000	Jump
IOT	6000	In/out transfer
OPR	7000	Operate



Memory Reference Instruction Bit Assignments

GROUP 1 OPERATE MICROINSTRUCTIONS

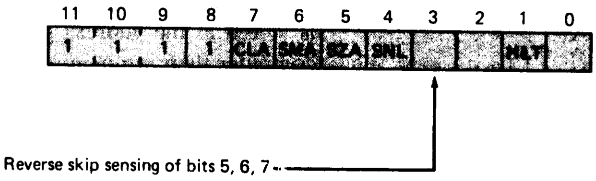
			Sequence
NOP	7000	No operation	–
CLA	7200	Clear AC	1
CLL	7100	Clear link	1
CMA	7040	Complement AC	2
CML	7020	Complement link	2
RAR	7010	Rotate AC and link right one	4
RAL	7004	Rotate AC and link left one	4
RTR	7012	Rotate AC and link right two	4
RTL	7006	Rotate AC and link left two	4
IAC	7001	Increment AC	3
BSW	7002	Swap bytes in AC	4



Logical sequences
1 CLA, CLL
2 CMA, CML
3 IAC
4 RAR, RAL, RTR, RTL, BSW

GROUP 2 OPERATE MICROINSTRUCTIONS

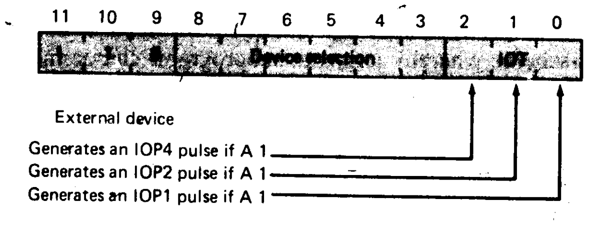
SMA	7500	Skip on minus AC
SZA	7440	Skip on zero AC
SPA	7510	Skip on plus AC
SNA	7450	Skip on nonzero AC
SNL	7420	Skip on nonzero link
SZL	7430	Skip on zero link
SKP	7410	Skip unconditionally
OSR	7404	Inclusive OR, switch register with AC
HLT	7402	Halts the program
CLA	7600	Clear AC



Logical sequences
1 (Bit 8 is zero) Either SMA or SZA or SZA or SNL
1 (Bit 8 is one) Both SPA and SNA and SZL
2 CLA
3 OSR, HLT

INTERNAL IOT MICROINSTRUCTIONS
PROGRAM INTERRUPT AND FLAG (1.2 μs)

SKON	6000	Skip if interrupt ON, and turn OFF
ION	6001	Turn interrupt ON
IOF	6002	Turn interrupt OFF
SRQ	6003	Skip on interrupt request
GTF	6004	Get interrupt flags
RTF	6005	Restore interrupt flags
SGT	6006	Skip on Greater Than flag
CAF	6007	Clear all flags


11 10 9 8 7 6 5 4 3 2 1 0
1
1
0
Device selection
IOT
External device
Generates an IOP4 pulse if A 1
Generates an IOP2 pulse if A 1
Generates an IOP1 pulse if A 1

IOT Instruction Bit Assignments

FIGURE 10.11

Instruction repertoire for the 6100. ***(Harris Semiconductor.)***

These subprograms or subroutines[6] are then jumped to whenever the function they perform is required [refer to Fig. 10.12(*a*)].

The problem confronting the computer designer is that a given subprogram can be jumped to from several different locations in the program. This is shown in Fig. 10.12(*b*). For instance, in minicomputers and microcomputers no square root instruction is provided. If many square roots are called for in a program, the programmer writes a single square root subprogram (or subroutine), and whenever the program must find a square root, a jump is made to this subprogram. After the square root has been formed, the subprogram then causes a jump back to the instruction following the JMS subprogram instruction in the original program section. The subprogram is said to be *called,* and it exits by returning to the *calling program*.

The problem is to arrange for a smooth jump to the called subprogram and to make it easy for the subprogram to return to the original program. To implement this, it is necessary to "plant" the address of the instruction to be returned to when the subprogram is finished in some convenient place for the subprogram. Since the program counter (instruction counter) contains this address when the jump is made, most computers provide a "jump to subroutine" instruction that will store the program counter before the jump is made.

The JMS instruction in the 6100 operates as follows. The program counter is stored at the address given in the address portion of the JMS instruction. The computer then jumps to the next address in memory. That is, if we write JMS 50_8 (where 50_8 is the 50th location in memory) and if our instruction word is at 201_8 in the memory, then when the JMS 50_8 instruction is executed, the value 202_8 will be stored at location 50_8 in the memory, and the computer will actually jump to or execute the word at location 51_8 next.[7]

Planting the program counter's contents at location 50_8 in the above example enables the writer of the subroutine to exit from the subprogram by placing a JMP (jump) instruction at the end of the subprogram, using address 50_8 in the address section but making the address an indirect address. Therefore, the computer will actually jump to the address stored at location 50_8, which will be 202_8, and the next instruction executed will be that at location 202_8.

Notice that a subprogram set up in this way can be jumped to by a JMS and exited by using a JMP indirectly from any place in memory, and the return will always be correct.

The above scheme is a good one and has been used (with variations) in several computers. But it has the problem that if a subprogram calls another subprogram which calls the first subprogram, then the return address for the original return will be wiped out. While this may seem unlikely, programs can become very complicated, and this must be avoided.

If a subprogram can call itself or can call another program which calls it without damage, then the subprogram is said to be *recursive*.

A scheme whereby jumps to subprograms can be made so that a subprogram can call itself is given in a later section.

[6] We use the words *subroutine* and *subprogram* to mean the same thing. Different manufacturers use different words.

[7] DEC uses octal numbers almost exclusively in 6100 programming.

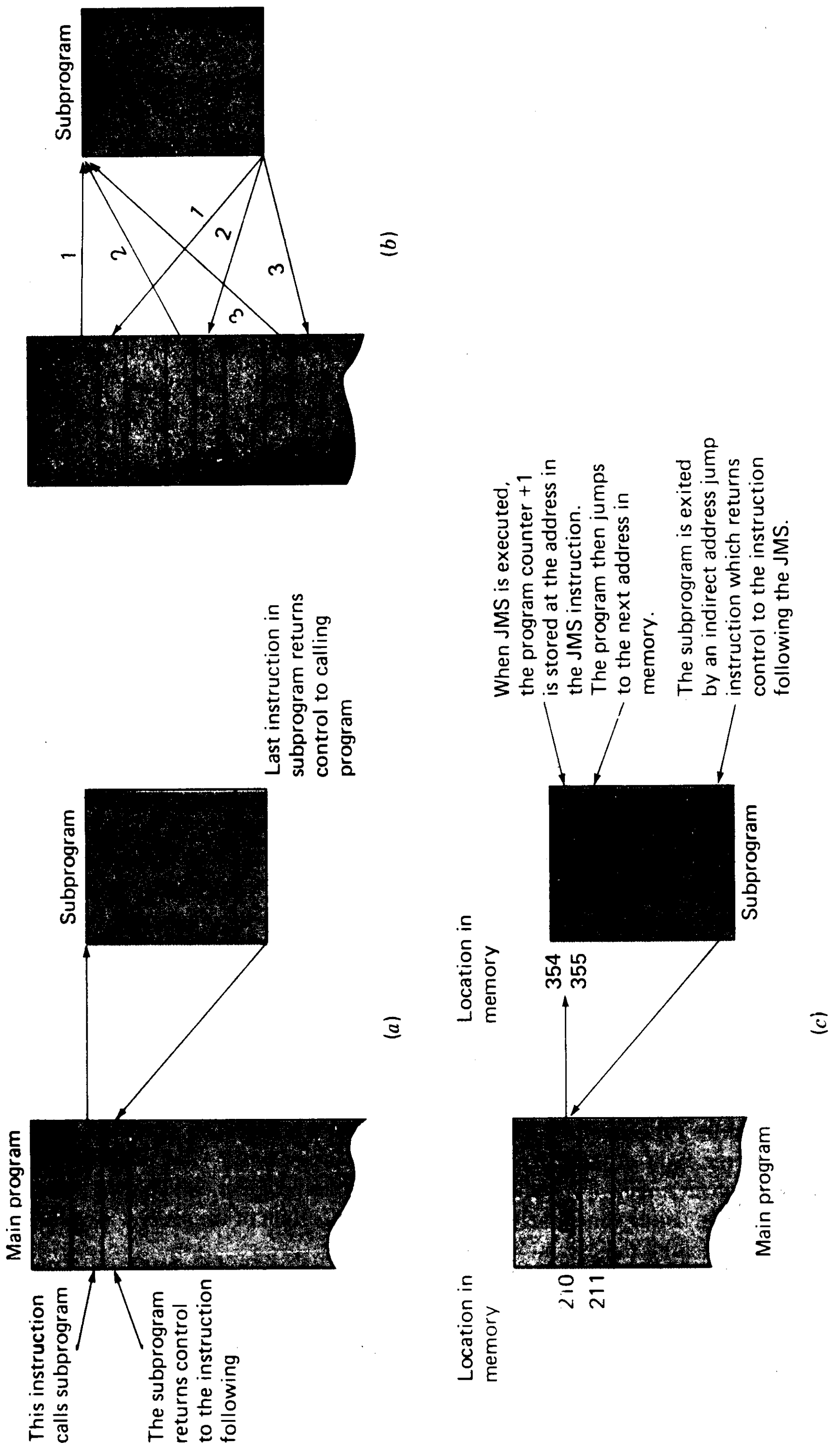

Subprogram
1
2
3
1
2
3
(b)
Main program
This instruction calls subprogram
The subprogram returns control to the instruction following
Subprogram
Last instruction in subprogram returns control to calling program
(a)
Location in memory
210
211
Main program
Location in memory
354
355
Subprogram
When JMS is executed, the program counter +1 is stored at the address in the JMS instruction. The program then jumps to the next address in memory.
The subprogram is exited by an indirect address jump instruction which returns control to the instruction following the JMS.
(c)

FIGURE 10.12

(*a*) Calling a subprogram. (*b*) The same subprogram may be called several times. (*c*) How the 6100 handles subprogram calls and returns.

The 6100 is bused, and input-output devices are connected to this bus. Input-output in the 6100 is provided by the IOT (input-output transfer) instruction. The input-output devices are each assigned a number from 0 to $2^6 - 1$. When an IOT instruction is given, the number in bits 3 to 8 of the IOT is placed on six wires in the bus which each input-output device inputs to see whether it is being addressed. The remaining 3 bits are also transmitted on the bus and tell the input-output device what to do (read, write, rewind, etc.). The selected input-output device responds to the bus signals generated by the IOT instruction, using logic circuits in its interface to interpret the instruction and to place data on the data section of the bus, read from it, etc.

The input-output devices are allowed to interrupt the processor and demand service during program operation, by using the computer's interrupt facility. The ION (interrupt on) instruction raises a bus wire called ION to the input-output devices, telling them it is their right to raise the INTERRUPT line on the bus and demand service.

If an input-output device raises its INTERRUPT line while the computer is operating a program, the address of the next instruction word which would normally be executed is placed in location 0 in the memory, and the next instruction executed is at location 1 in the memory. (This is generally a jump to the subprogram which services input-output devices.) Since the address of the next instruction word in the program which was interrupted is in location 0, the interrupt service program[8] can exit, using that address to return to the original program. This is shown in Fig. 10.13.

Again, an interrupt of an interrupt will cause the original return address (at 0 in the memory) to be destroyed. The programmer must see that this does not occur, and the right of devices to interrupt is revoked as soon as an interrupt occurs. (The bus line for ION is lowered.) The program must issue another ION to restore the interrupt privilege to input-output devices.

The 6100 has a number of features which have helped to make it attractive, but which are outside the context of this description. For instance, since there is only a single arithmetic instruction, TAD, it is necessary to complement an operand and then add in order to subtract. To perform this complement and to provide ROTATE or SHIFT instructions, some SKIP instructions, and some other logic operations, the OP code 111 is a "no address" instruction class where the remaining 9 bits tell which of a number of different possible operations can be made to occur.

Table 10.2 shows a section of an assembler listing[9] for a 6100. The leftmost two columns (or digits) list the addresses in memory and their contents. The columns to the right of these were written by the programmer. The programmer fed the assembler-language statements to the assembler program, which then generated the complete listing shown here. All material to the right of the slashes is comments and is ignored by the assembler program.

The purpose of this subroutine, or subprogram, is to read from a keyboard

[8] The program (or subprogram) which handles the peripheral that generates the interrupt is called an *interrupt service program*.

[9] This listing is in octal. Each digit is an octal digit.

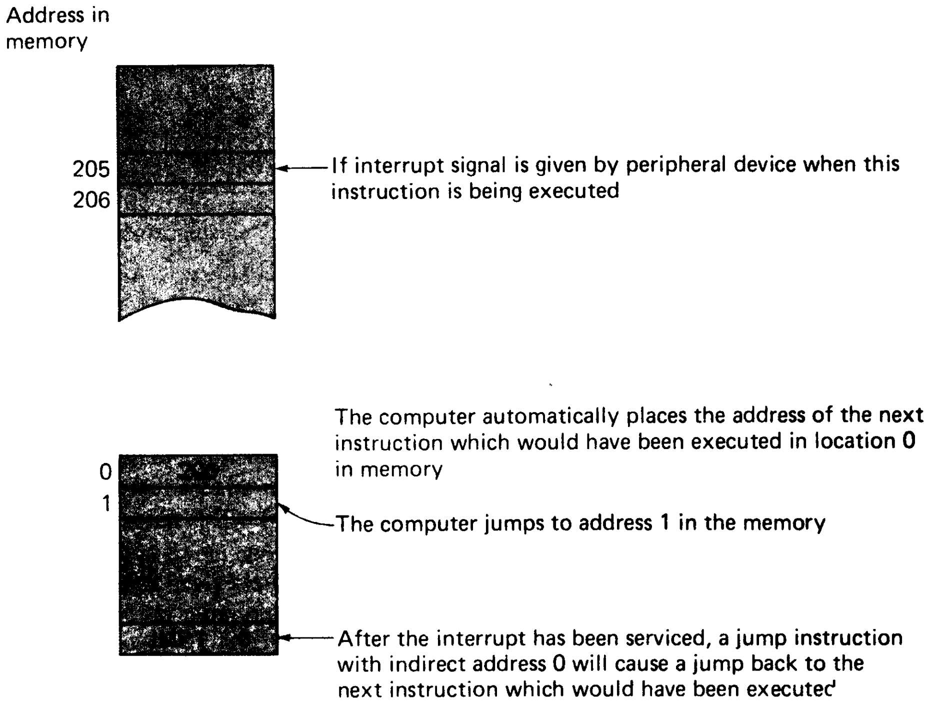

Address in memory
205
206
If interrupt signal is given by peripheral device when this instruction is being executed
0
1
The computer automatically places the address of the next instruction which would have been executed in location 0 in memory
The computer jumps to address 1 in the memory
After the interrupt has been serviced, a jump instruction with indirect address 0 will cause a jump back to the next instruction which would have been executed

FIGURE 10.13

Interrupts in the 6100 microprocessor.

into the 6100's accumulator. Another program enters this subprogram with a JMS statement, which deposits the address of the next instruction to be operated when the subroutine is completed in the first address of the subprogram. The address to be returned to will be stored at location 251 in the memory when the subroutine is entered, and the first statement executed will be the statement at location 252 in the memory.

The statement at 252 in the memory is a KSF statement, which is a special input statement that reads from the keyboard's status register (see Chap. 7 for details). If the status bit is a 1 in this register, the program skips over the next instruction. As a result, when the KSF statement is executed, if there is a character to be read from the keyboard, the next statement read will be at location 254. If the status word is all 0s, then the statement executed after the KSF will be the JMP instruction at 253 in the memory.

TABLE 10.2 **6100 SUBPROGRAM**

ADDRESS	CONTENTS	LABEL	OP CODE	COMMENTS
Ø251	ØØØØ	LISN,	Ø	/INPUT SUB
Ø252	6Ø31		KSF	
Ø253	5252		JMP #252	
Ø254	6Ø36		KRB	
Ø255	5651		JMP I LISN	

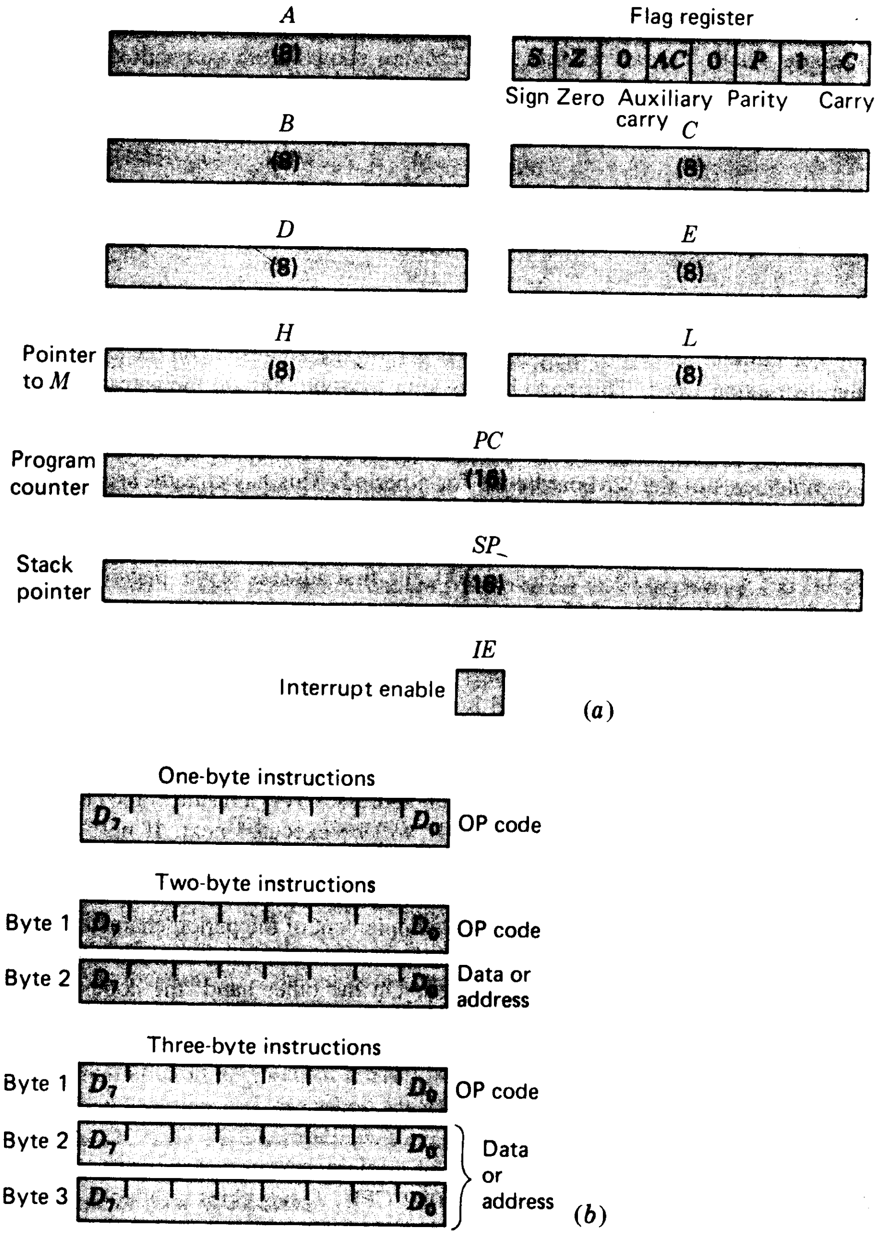


FIGURE 10.14

(*a*) 8080 CPU registers. (*b*) Instruction word formats for 8080 CPU.

The 8080 CPU is basically a single-accumulator organization, but a number of other scratchpad registers are provided, as illustrated in Fig. 10.14(*a*). The instruction word formats are shown in Fig. 10.14(*b*), and a functional block diagram of the 8080 is shown in Fig. 10.15.

The Intel 8080 has a good set of addressing modes (see Table 10.4). At the beginning of each instruction cycle, the 8-bit OP code is read by the 8080 CPU, and this determines how many more fetches from memory the CPU must make to

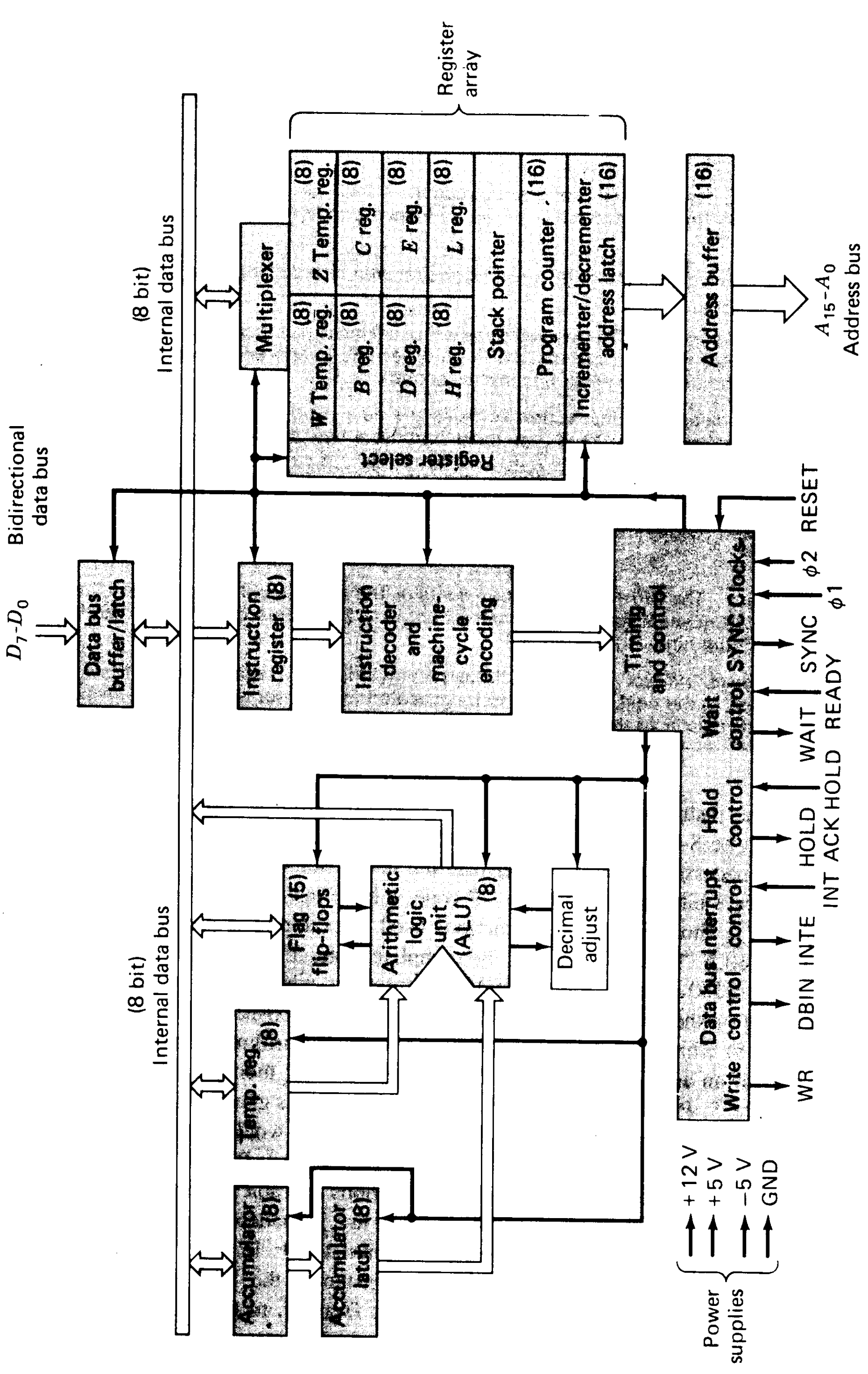


FIGURE 10.15

8080 CPU functional block diagram. (*Intel Corp.*)

TABLE 10.4 ADDRESSING MODES FOR THE 8080 CPU

When multibyte numeric data are used, the data, like instructions, are stored in successive memory locations, with the least significant byte first, followed by increasingly significant bytes. The 8080 has four different modes for addressing data stored in memory or in registers:

1 *Direct* Bytes 2 and 3 of the instruction contain the exact memory address of the data item (the low-order bits of the address are in byte 2, the high-order bits in byte 3).

2 *Register* The instruction specifies the register or register pair in which the data are located.

3 *Register indirect* The instruction specifies a register pair that contains the memory address where the data are located (high-order bits of the address are in the first register of the pair, low-order bits in the second).

4 *Immediate* The instruction contains the data. This is either an 8- or a 16-bit quantity (least significant byte first, most significant byte second).

Unless directed by an INTERRUPT or a BRANCH instruction, the execution of instructions proceeds through consecutively increasing memory locations. A BRANCH instruction can specify the address of the next instruction to be executed in one of two ways:

1 *Direct* The BRANCH instruction contains the address of the next instruction to be executed. (Except for the RST instruction, byte 2 contains the low-order address and byte 3 the high-order address.)

2 *Register indirect* The BRANCH instruction indicates a register pair that contains the address of the next instruction to be executed (high-order bits of the address are in the first register of the pair, low-order bits in the second).

execute the instruction. Some instructions require only the 8-bit OP code, while others require 8-bit and some 16-bit addresses or operands, and so the CPU must make the necessary accesses to perform the instruction.

Since 8 bits are used for the OP code, a large instruction repertoire has been provided. A short list of the instructions used in examples is presented in Tables 10.5 and 10.6. Table 10.7 shows the complete instruction set.

The 8080 has conditional JUMP instructions which jump or do not jump, depending on the values in the *condition flags*. These consist of five flip-flops (see Table 10.8) which are set to 0 or 1 by the results of arithmetic instructions. For instance, if an addition is performed and the result is 0, then the *Z* flag will be set to 1 and the *S*, *P*, and *C* flags to 0 (provided no carry was generated). A conditional JUMP instruction which tests the *Z* flag for a 1 state would then cause a jump, whereas a conditional JUMP instruction which tests the *S*, *P*, or *C* flags would not cause a jump. (Refer to the BRANCH instructions in Table 10.6.)

In programming the 8080, considerable use is made of the scratchpad registers *B*, *C*, *D*, *E*, *H*, and *L* as well as accumulator *A*. In some instructions the scratchpad registers are used in pairs. For instance, the INR M instruction, a 1-byte instruction, uses the two 8-bit registers *H* and *L* to form a 16-bit address. The 8-bit number in the memory at this address is then incremented by the instruction. The *Z*, *S*, *P*, and *AC* flags are all set and reset by the instruction, so a 0 result at that location will set the *Z* flag, a negative result will set the *S* flag, etc.

TABLE 10.5 NOTATION FOR 8080 CPU INSTRUCTION REPERTOIRE LISTING IN TABLE 10.6

SYMBOL	MEANING
Accumulator	Register *A*
Addr	16-bit address quantity
Data	8-bit data quantity
Data 16	16-bit data quantity
Byte 2	Second byte of the instruction
Byte 3	Third byte of the instruction
Port	8-bit address of an input-output device.
r, r1, r2	One of the registers *A, B, C, D, E, H, L*
DDD, SSS	Bit pattern designating one of the registers *A, B, C, D, E, H, L* (DDD = destination, SSS = source): DDD or SSS — Register Name 111 — *A* 000 — *B* 001 — *C* 010 — *D* 011 — *E* 100 — *H* 101 — *L*
rh	First (high-order) register of a designated register pair
rl	Second (low-order) register of a designated register pair
rp	One of the register pairs: *B* represents the *B, C* pair with *B* as the high-order register and *C* as the low-order register. *D* represents the *D, E* pair with *D* as the high-order register and *E* as the low-order register. *H* represents the *H, L* pair with *H* as the high-order register and *L* as the low-order register. *SP* represents the 16-bit stack pointer register.
RP	Bit pattern designating one of the register pairs *B, D, H, SP*: RP — Register pair 00 — *B, C* 01 — *D, E* 10 — *H, L* 11 — *SP*
PC	16-bit program counter register (PCH and PCL are used to refer to the high-order and low-order 8 bits, respectively)
SP	Stack pointer
r_m	Bit *m* of register *r* (bits are numbered 7 through 0 from left to right)
Z, S, P, CY, AC	Condition flags: zero, sign, parity, carry, and auxiliary carry, respectively.
( )	Contents of memory location or registers enclosed in the parentheses
←	"Is transferred to"
∧	Logical AND
⊻	Exclusive OR
∨	Inclusive OR
+	Addition
−	2s complement subtraction
*	Multiplication
↔	"Is exchanged with"
—	Is complement [e.g., $\overline{(A)}$]
n	Restart numbers 0 to 7
NNN	Binary representation 000 to 111 for restart numbers 0 to 7, respectively

TABLE 10.6 INSTRUCTION REPERTOIRE FOR 8080 CPU

MOV r1, r2 (move register)
(r1) ← (r2)
Content of register r2 is moved to register r1.

0	1	D	D	D	S	S	S

Addressing: register Flags: none

MOV r, M (move from memory)
(r) ← ((H)(L))
Content of memory location, whose address is in registers *H* and *L*, is moved to register *r*.

0	1	D	D	D	1	1	0

Addressing: register indirect Flags: none

MOV M, r (move to memory)
((H)(L)) ← (r)
Content of register r is moved to memory location whose address is in registers *H* and *L*.

0	1	1	1	0	S	S	S

Addressing: register indirect Flags: none

ADD r (add register)
(A) ← (A) + (r)
Content of register *r* is added to content of accumulator. Result is placed in accumulator.

1	0	0	0	0	S	S	S

Addressing: register Flags: *Z, S, P, CY, AC*

ADD M (add memory)
(A) ← (A) + ((H)(L))
Content of the memory location, whose address is contained in registers *H* and *L*, is added to content of accumulator. Result is placed in accumulator.

1	0	0	0	0	1	1	0

Addressing: register indirect
Flags: *Z, S, P, CY, AC*

ADI data (add immediate)
(A) ← (A) + (byte 2)
Content of second byte of instruction is added to content of accumulator. Result is placed in accumulator.

<table>
<tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">Data</td></tr>
</table>

Addressing: immediate
Flags: *Z, S, P, CX, AC*

INR M (increment memory)
((H)(L)) ← ((H)(L)) + 1
Content of the memory location, whose address is contained in registers *H* and *L* is incremented by 1. *Note:* All condition flags except *CY* are affected.

0	0	1	1	0	1	0	0

Addressing: register indirect
Flags: *Z, S, P, AC*

LDA addr (load accumulator direct)
(A) ← ((byte 3)(byte 2))
Content of memory location, whose address is specified in byte 2 and byte 3 of instruction, is moved to register *A*.

<table>
<tr><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">Low-order addr</td></tr>
<tr><td colspan="8">High-order addr</td></tr>
</table>

Addressing: direct Flags: none

STA addr (store accumulator direct)
((byte 3)(byte 2)) ← (A)
Content of accumulator is moved to memory location whose address is specified in bytes 2 and 3 of instruction.

<table>
<tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">Low-order</td></tr>
<tr><td colspan="8">High-order addr</td></tr>
</table>

Addressing: direct Flags: none

SUB M (subtract memory)
(A) ← (A) − ((H)(L))
Content of memory location, whose address is contained in registers *H* and *L*, is subtracted from content of accumulator. Result is placed in accumulator.

1	0	0	1	0	1	1	0

Addressing: register indirect
Flags: *Z, S, P, CY, AC*

SUI data (subtract immediate)
(A) ← (A) − (byte 2)
Content of second byte of instruction is subtracted from content of accumulator. Result is placed in accumulator.

TABLE 10.6 INSTRUCTION REPERTOIRE FOR 8080 CPU (*continued*)

1	1	0	1	0	1	1	0
Data							

Addressing: immediate
Flags: *Z, S, P, CY, AC*

SBB r (subtract register with borrow)
(A) ← (A) − (r) − (CY)

Content of register *r* and content of *CY* flag are both subtracted from accumulator. Result is placed in accumulator.

1	0	0	1	1	S	S	S

Addressing: register Flags: *Z, S, P, CY, AC*

CMP M (compare memory)
(A) − ((H)(L))

Content of memory location, whose address is contained in registers *H* and *L*, is subtracted from accumulator. Accumulator remains unchanged. Condition flags are set as a result of subtraction. *Z* flag is set to 1 if (A) = ((H)(L)). *CY* flag is set to 1 if (A) < ((H)(L)).

1	0	1	1	1	1	1	0

Cycles: 2 States: 7
Addressing: register indirect
Flags: *Z, S, P, CY, AC*

INX rp (increment register pair)
(rh)(rl) ← (rh)(rl) + 1

Content of register pair *rp* is incremented by 1. *Note:* No condition flags are affected.

0	0	R	P	0	0	1	1

Cycles: 1 States: 5
Addressing: register Flags: none

DCX rp (decrement register pair)
(rh)(rl) ← (rh)(rl) − 1

Content of register pair *rp* is decremented by 1. *Note:* No condition flags are affected.

0	0	R	P	1	0	1	1

Cycles: 1 States: 5
Addressing: register Flags: none

RST n (restart)
((SP) − 1) ← (PCH) ((SP) − 2) ← (PCL)
(SP) ← (SP) − 2 (PC) ← 8* (NNN)

The high-order 8 bits of next instruction address are moved to memory location whose address is 1 less than content of register *SP*. The low-order 8 bits of next instruction address are moved to memory location whose address is 2 less than content of register *SP*. Content of register *SP* is decremented by 2. Control is transferred to instruction whose address is eight times the content of NNN.

1	1	N	N	N	1	1	1

Cycles: 3 States: 11
Addressing: register indirect Flags: none

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	N	N	N	0	0	0

Program counter after restart

LXI rp, data 16 (load register pair immediate)
(rh) ← (byte 3) (rl) ← (byte 2)

Byte 3 of instruction is moved into high-order register *rh* of register pair *rp*. Byte 2 of instruction is moved into low-order register *rl* of register pair *rp*.

0	0	R	P	0	0	0	1
Low-order data							
High-order data							

Addressing: immediate Flags: none

BRANCH GROUP

This group of instructions alters normal sequential program flow. *Condition flags are not affected by any instruction in this group.*

The two types of branch instructions are unconditional and conditional. Unconditional transfers simply perform the specified operation on register *PC* (the program counter). Conditional transfers examine the status of one of the four processor flags to determine whether the specified branch is to be executed. The conditions that may be specified are as follows:

CCC		Condition
000	NZ	Not zero (*Z* = 0)
001	Z	Zero (*Z* = 1)
010	NC	No carry (*CY* = 0)
011	C	Carry (*CY* = 1)
100	PO	Parity odd (*P* = 0)
101	PE	Parity even (*P* = 1)
110	P	Plus (*S* = 0)
111	M	Minus (*S* = 1)

JMP addr (jump)
(PC) ← (byte 3)(byte 2)

Control is transferred to instruction whose address is specified in byte 3 and byte 2 of current instruction.

TABLE 10.6 INSTRUCTION REPERTOIRE FOR 8080 CPU (*continued*)

<table>
<tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td colspan="8">Low-order addr</td></tr>
<tr><td colspan="8">High-order addr</td></tr>
</table>

Addressing: immediate   Flags: none

Jcondition addr (conditional jump)
If (CCC), (PC) ← (byte 3)(byte 2)
If specified condition is true, control is transferred to instruction whose address is specified in byte 3 and byte 2 of current instuction; otherwise, control continues sequentially.

<table>
<tr><td>1</td><td>1</td><td>C</td><td>C</td><td>C</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">Low-order addr</td></tr>
<tr><td colspan="8">High-order addr</td></tr>
</table>

Addressing: immediate   Flags: none

DCR r (decrement register)
(r) ← (r) − 1
Content of register *r* is decremented by 1. *Note:* All condition flags except *CY* are affected.

0	0	D	D	D	1	0	1

Addressing: register   Flags: *Z, S, P, AC*

LDAX rp (load accumulator indirect)
(A) ← ((rp))
Content of memory location, whose address is in register pair *rp*, is moved to register *A*. *Note:* Only register pairs *rp* = *B* (registers *B* and *C*) or *rp* = *D* (registers *D* and *E*) may be specified.

0	0	R	P	1	0	1	0

Addressing: register indirect   Flags: none

STAX rp (store accumulator indirect)
((rp)) ← (A)
Content of register *A* is moved to memory location whose address is in register pair *rp*. *Note:* Only register pairs *rp* = *B* (registers *B* and *C*) or *rp* = *D* (registers *D* and *E*) may be specified.

0	0	R	P	0	0	1	0

Addressing: register indirect   Flags: none

XCHG (exchange H and L with D and E)
(H) ↔ (D)   (L) ↔ (E)
Contents of registers *H* and *L* are exchanged with contents of registers *D* and *E*.

1	1	1	0	1	0	1	1

Addressing: register   Flags: none

INR r (increment register)
(r) ← (r) + 1
Content of register *r* is incremented by 1. *Note:* All condition flags except *CY* are effected.

0	0	D	D	D	1	0	0

Addressing: register   Flags: *Z, S, P, AC*

MVI M, data (move to memory immediate)
((H)(L)) ← (byte 2)
Content of byte 2 of instruction is moved to memory location whose address is in registers *H* and *L*.

<table>
<tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">Data</td></tr>
</table>

Addressing: immediate register indirect
Flags: none

CALL addr (call)
((SP) − 1) ← (PCH)   (SP) ← (SP) − 2
((SP) − 2) ← (PCL)   (PC) ← (byte 3)(byte 2)
The high-order 8 bits of next instruction address are moved to memory location whose address is 1 less than content of register *SP*. The low-order 8 bits of next instruction address are moved to memory location whose address is 2 less than content of register *SP*. Content of register *SP* is decremented by 2. Control is transferred to instruction whose address is specified in byte 3 and byte 2 of current instruction.

<table>
<tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td colspan="8">Low-order addr</td></tr>
<tr><td colspan="8">High-order addr</td></tr>
</table>

Addressing: immediate/register indirect
Flags: none

RET (return)
(PCL) ← ((SP))   (PCH) ← ((SP) + 1))
(SP) ← (SP) + 2
Content of memory location, whose address is specified in register *SP*, is moved to low-order 8 bits of register *PC*. Content of memory location, whose address is 1 more than content of register *SP*, is moved

TABLE 10.6 INSTRUCTION REPERTOIRE FOR 8080 CPU (*continued*)

to high-order 8 bits of register *PC*. Content of register *SP* is incremented by 2.

1	1	0	0	1	0	0	1

Addressing: register indirect Flags: none

STACK, INPUT-OUTPUT, AND MACHINE CONTROL GROUP

This group of instructions performs input-output, manipulates the stack, and alters internal control flags.

Unless otherwise specified, *condition flags are not affected by any instruction in this group.*

PUSH rp (push)
((SP) − 1) ← (rh) ((SP) − 2) ← (rl)
(SP) ← (SP) − 2
Content of high-order register of register pair *rp* is moved to memory location whose address is 1 less than content of register *SP*. Content of low-order register of register pair *rp* is moved to memory location whose address is 2 less than content of register *SP*. Content of register *SP* is decremented by 2. *Note:* Register pair *rp* = *SP* may not be specified.

1	1	R	P	0	1	0	1

Addressing: register indirect Flags: none

PUSH PSW (push processor status word)
((SP) − 1) ← (A)
$((SP) - 2)_0 \leftarrow (CY)$, $((SP) - 2)_1 \leftarrow 1$
$((SP) - 2)_2 \leftarrow (P)$, $((SP) - 2)_3 \leftarrow 0$
$((SP) - 2)_4 \leftarrow (AC)$, $((SP) - 2)_5 \leftarrow 0$
$((SP) - 2)_6 \leftarrow (Z)$, $((SP) - 2)_7 \leftarrow (S)$
(SP) ← (SP) − 2
Content of register *A* is moved to memory location whose address is 1 less than register *SP*. Contents of condition flags are assembled into a processor status word, and word is moved to memory location whose address is 2 less than content of register *SP*. Content of register *SP* is decremented by 2.

1	1	1	1	0	1	0	1

Addressing: register indirect Flags: none

POP rp (pop)
(rl) ← ((SP)) (rh) ← ((SP) + 1)
(SP) ← (SP) + 2
Content of memory location, whose address is specified by content of register *SP*, is moved to low-order register of register pair *rp*. Content of memory location, whose address is 1 more than content of register *SP*, is moved to high-order register of register pair *rp*. Content of register *SP* is incremented by 2. *Note:* Register pair *rp* = *SP* may not be specified.

1	1	R	P	0	0	0	1

Addressing: register indirect Flags: none

POP PSW (pop processor status word)
$(CY) \leftarrow ((SP))_0$ $(P) \leftarrow ((SP))_2$ $(AC) \leftarrow ((SP))_4$
$(Z) \leftarrow ((SP))_6$ $(S) \leftarrow ((SP))_7$ $(A) \leftarrow ((SP + 1))$
(SP) ← (SP) + 2
Content of memory location, whose address is specified by content of register *SP*, is used to restore condition flags. Content of memory location, whose address is 1 more than content of register *SP*, is moved to register *A*. Content of register *SP* is incremented by 2.

1	1	1	1	0	0	0	1

Addressing: register indirect
Flags: *Z, S, P, CY, AC*

Parentheses are used to indicate "the contents of" in Tables 10.5 and 10.6. For example, the notation (H) ↔ (D) used in the XCHG instruction means, "The contents of register *H* and register *D* are exchanged." Similarly, ((H)(L)) ← (byte 2) in the description of the MVI instruction means, "The contents of byte 2 of the instruction word are transferred into the location in memory whose address is formed by writing the contents of register *H* to the left of the contents of register *L*." This notation is widely used and worth examining in some detail.

Table 10.9 shows an 8080 program in assembly language and a listing of the hexadecimal values for memory location and contents as generated by the assembler. The programmer wrote the columns: Label, OP Code, Operand, and Comments. The assembler generated the two leftmost columns.

TABLE 10.7 8080 INSTRUCTION REPERTOIRE

MNEMONIC	DESCRIPTION	INSTRUCTION CODE†							
		D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
MOV r1, r2	Move register to register	0	1	D	D	D	S	S	S
MOV M, r	Move register to memory	0	1	1	1	0	S	S	S
MOV r, M	Move memory to register	0	1	D	D	D	1	1	0
HLT	Halt	0	1	1	1	0	1	1	0
MVI r	Move immediate register	0	0	D	D	D	1	1	0
MVI M	Move immediate memory	0	0	1	1	0	1	1	0
INR r	Increment register	0	0	D	D	D	1	0	0
DCR r	Decrement register	0	0	D	D	D	1	0	1
INR M	Increment memory	0	0	1	1	0	1	0	0
DCR M	Decrement memory	0	0	1	1	0	1	0	1
ADD r	Add register to *A*	1	0	0	0	0	S	S	S
ADC r	Add register to *A* with carry	1	0	0	0	1	S	S	S
SUB r	Subtract register from *A*	1	0	0	1	0	S	S	S
SBB r	Subtract register from *A* with borrow	1	0	0	1	1	S	S	S
ANA r	AND register with *A*	1	0	1	0	0	S	S	S
XRA r	Exclusive-OR register with *A*	1	0	1	0	1	S	S	S
ORA r	OR register with *A*	1	0	1	1	0	S	S	S
CMP r	Compare register with *A*	1	0	1	1	1	S	S	S
ADD M	Add memory to *A*	1	0	0	0	0	1	1	0
ADC M	Add memory to *A* with carry	1	0	0	0	1	1	1	0
SUB M	Subtract memory from *A*	1	0	0	1	0	1	1	0
SBB M	Subtract memory from *A* with borrow	1	0	0	1	1	1	1	0
ANA M	AND memory with *A*	1	0	1	0	0	1	1	0
XRA M	Exclusive-OR memory with *A*	1	0	1	0	1	1	1	0
ORA M	OR memory with *A*	1	0	1	1	0	1	1	0
CMP M	Compare memory with *A*	1	0	1	1	1	1	1	0
ADI	Add immediate to *A*	1	1	0	0	0	1	1	0
ACI	Add immediate to *A* with carry	1	1	0	0	1	1	1	0
SUI	Subtract immediate from *A*	1	1	0	1	0	1	1	0
SBI	Subtract immediate from *A* with borrow	1	1	0	1	1	1	1	0
ANI	AND immediate with *A*	1	1	1	0	0	1	1	0
XRI	Exclusive-OR immediate with *A*	1	1	1	0	1	1	1	0
ORI	OR immediate with *A*	1	1	1	1	0	1	1	0
CPI	Compare immediate with *A*	1	1	1	1	1	1	1	0
RLC	Rotate *A* left	0	0	0	0	0	1	1	1
RRC	Rotate *A* right	0	0	0	0	1	1	1	1
RAL	Rotate *A* left through carry	0	0	0	1	0	1	1	1
RAR	Rotate *A* right through carry	0	0	0	1	1	1	1	1
JMP	Jump unconditional	1	1	0	0	0	0	1	1
JC	Jump on carry	1	1	0	1	1	0	1	0
JNC	Jump on no carry	1	1	0	1	0	0	1	0
JZ	Jump on zero	1	1	0	0	1	0	1	0
JNZ	Jump on no zero	1	1	0	0	0	0	1	0
JP	Jump on positive	1	1	1	1	0	0	1	0
JM	Jump on minus	1	1	1	1	1	0	1	0
JPE	Jump on parity even	1	1	1	0	1	0	1	0
JPO	Jump on parity odd	1	1	1	0	0	0	1	0
CALL	Call unconditional	1	1	0	0	1	1	0	1
CC	Call on carry	1	1	0	1	1	1	0	0
CNC	Call on no carry	1	1	0	1	0	1	0	0
CZ	Call on zero	1	1	0	0	1	1	0	0
CNZ	Call on no zero	1	1	0	0	0	1	0	0
CP	Call on positive	1	1	1	1	0	1	0	0
CM	Call on minus	1	1	1	1	1	1	0	0
CPE	Call on parity even	1	1	1	0	1	1	0	0
CPO	Call on parity odd	1	1	1	0	0	1	0	0

TABLE 10.7 8080 INSTRUCTION REPERTOIRE (*continued*)

		INSTRUCTION CODE†							
MNEMONIC	DESCRIPTION	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
RET	Return	1	1	0	0	1	0	0	1
RC	Return on carry	1	1	0	1	1	0	0	0
RNC	Return on no carry	1	1	0	1	0	0	0	0
RZ	Return on zero	1	1	0	0	1	0	0	0
RNZ	Return on no zero	1	1	0	0	0	0	0	0
RP	Return on positive	1	1	1	1	0	0	0	0
RM	Return on minus	1	1	1	1	1	0	0	0
RPE	Return on parity even	1	1	1	0	1	0	0	0
RPO	Return on parity odd	1	1	1	0	0	0	0	0
RST	Restart	1	1	A	A	A	1	1	1
IN	Input	1	1	0	1	1	0	1	1
OUT	Output	1	1	0	1	0	0	1	1
LXI B	Load imediate register pair *B, C*	0	0	0	0	0	0	0	1
LXI D	Load immediate register pair *D, E*	0	0	0	1	0	0	0	1
LXI H	Load immediate register pair *H, L*	0	0	1	0	0	0	0	1
LXI SP	Load immediate stack pointer	0	0	1	1	0	0	0	1
PUSH B	Push register pair *B, C* on stack	1	1	0	0	0	1	0	1
PUSH D	Push register pair *D, E* on stack	1	1	0	1	0	1	0	1
PUSH H	Push register pair *H, L* on stack	1	1	1	0	0	1	0	1
PUSH PSW	Push *A* and flags on stack	1	1	1	1	0	1	0	1
POP B	Pop register pair *B, C* off stack	1	1	0	0	0	0	0	1
POP D	Pop register pair *D, E* off stack	1	1	0	1	0	0	0	1
POP H	Pop register pair *H, L* off stack	1	1	1	0	0	0	0	1
POP PSW	Pop *A* and flags off stack	1	1	1	1	0	0	0	1
STA	Store *A* direct	0	0	1	1	0	0	1	0
LDA	Load *A* direct	0	0	1	1	1	0	1	0
XCHG	Exchange *D, E* and *H, L* registers	1	1	1	0	1	0	1	1
XTHL	Exchange top of stack, *H, L*	1	1	1	0	0	0	1	1
SPHL	*H, L* to stack pointer	1	1	1	1	1	0	0	1
PCHL	*H, L* to program counter	1	1	1	0	1	0	0	1
DAD B	Add *B, C* to *H, L*	0	0	0	0	1	0	0	1
DAD D	Add *D, E* to *H, L*	0	0	0	1	1	0	0	1
DAD H	Add *H, L* to *H, L*	0	0	1	0	1	0	0	1
DAD SP	Add stack pointer to *H, L*	0	0	1	1	1	0	0	1
STAX B	Store *A* indirect	0	0	0	0	0	0	1	0
STAX D	Store *A* indirect	0	0	0	1	0	0	1	0
LDAX B	Load *A* indirect	0	0	0	0	1	0	1	0
LDAX D	Load *A* indirect	0	0	0	1	1	0	1	0
INX B	Increment *B, C* registers	0	0	0	0	0	0	1	1
INX D	Increment *D, E* registers	0	0	0	1	0	0	1	1
INX H	Increment *H, L* registers	0	0	1	0	0	0	1	1
INX SP	Increment stack pointer	0	0	1	1	0	0	1	1
DCX B	Decrement *B, C*	0	0	0	0	1	0	1	1
DCX D	Decrement *D, E*	0	0	0	1	1	0	1	1
DCX H	Decrement *H, L*	0	0	1	0	1	0	1	1
DCX SP	Decrement stack pointer	0	0	1	1	1	0	1	1
CMA	Complement *A*	0	0	1	0	1	1	1	1
STC	Set carry	0	0	1	1	0	1	1	1
CMC	Complement carry	0	0	1	1	1	1	1	1
DAA	Decimal adjust *A*	0	0	1	0	0	1	1	1
SHLD	Store *H, L* direct	0	0	1	0	0	0	1	0
LHLD	Load *H, L* direct	0	0	1	0	1	0	1	0
EI	Enable interrupt	1	1	1	1	1	0	1	1
DI	Disable interrupt	1	1	1	1	0	0	1	1
NOP	No operation	0	0	0	0	0	0	0	0

Note: DDD or SSS is numbered as follows: 000, B; 001, C; 010, D; 011, E; 100, H; 101, L; 110, memory; 111, A. For example, 01010001 instructs the computer to move the contents of register *C* into register *D*.

TABLE 10.8 FLAGS USED IN 8080 CPU

Flag word

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
S	Z	0	AC	0	P	1	CY

[illegible] five condition flags associated with the execution of instructions on the [illegible] zero, sign, parity, carry, and auxiliary carry. Each is represented by a 1-bit register [illegible] CPU. A flag is "set" by forcing the bit to 1 and "reset" by forcing the bit to 0.

[illegible] When an instruction affects a flag, it affects it in the [illegible]

[illegible] Zero: If the result of an instruction has the value 0, this flag is set; otherwise, [illegible]

[illegible] bit of the result of the operation has the value 1, [illegible] set; otherwise, it is reset.

[illegible] bits of the result of the operation is 0 (i.e., [illegible] this flag is set; otherwise, it is reset (i.e., if the result has [illegible]

[illegible] resulted in a carry (from addition) or a borrow (from [illegible] out of the high-order bit, this flag is set; otherwise, it is [illegible]

[illegible] caused a carry out of bit 3 and into bit 4 of [illegible] carry is set; otherwise, it is reset. This flag is affected [illegible] increments, decrements, comparisons, and [illegible]

The purpose of the program is to find the larger of two 8-bit numbers in locations 50 and 51 in the memory and to store this number at location 52.

The first instruction, LXI H, 50H, loads 50_{16} into registers H and L. When the 8080 assembler is used, writing an H to the right of a number means the number is hexadecimal. Therefore 50H means 01010000_2, or 50_{16} to the assembler. So LXI loads 00000000 into register H and 01010000 into register L. (Notice that the least significant byte is first in the memory in an instruction word in the 8080.)

The MOV A, M instruction[11] moves the byte in the memory pointed to by the address in registers H and L into accumulator A. Since H and L point to location 50, the byte at that location will be moved into the accumulator.

The INX H instruction adds 1 to the register pair H, L, giving 51 in H and L.

The CMP M instruction compares the byte in the memory pointed to by the H, L pair with the contents of accumulator A and sets the status flags accordingly. In effect, the flags are set as if the byte in memory had been subtracted from accumulator A. However, neither memory nor accumulator is changed. As a result, if the byte in the memory equals that in A, the Z bit will be set to 1; if A is less than the byte in the memory, the C flag will be set to 1.

The JNC FINIS instruction causes a jump to FINIS if the C flag is a 0. (In this case the content of A is larger than or equal to that in location 51 in the

[11] M is used in assembler language to indicate the byte in the memory pointed to by the H, L pair of registers. These must have been properly set before such an instruction is used.

TABLE 10.9 8080 PROGRAM TO FIND LARGEST NUMBER

MEMORY ADDRESS	CONTENTS	LABEL	OP CODE	OPERAND	COMMENTS
[illegible]	[illegible]		[illegible]	H, 50H	LOAD H AND L
[illegible]	[illegible]		[illegible]	A, M	GET 1ST OPERAND
[illegible]	[illegible]		[illegible]	M	IS 2D OPERAND LARGER
[illegible]	[illegible]	[illegible]	[illegible]	A, M	2D OPERAND IS LARGER

memory.) If no jump is taken, the MOV A, M instruction moves the byte at location 51 (now pointed to by *H, L*) into the accumulator.

The INX H instruction adds 1 to the *H, L* register pair, giving 52, and the MOV M, A instruction moves the contents of the accumulator into location 52 in the memory.

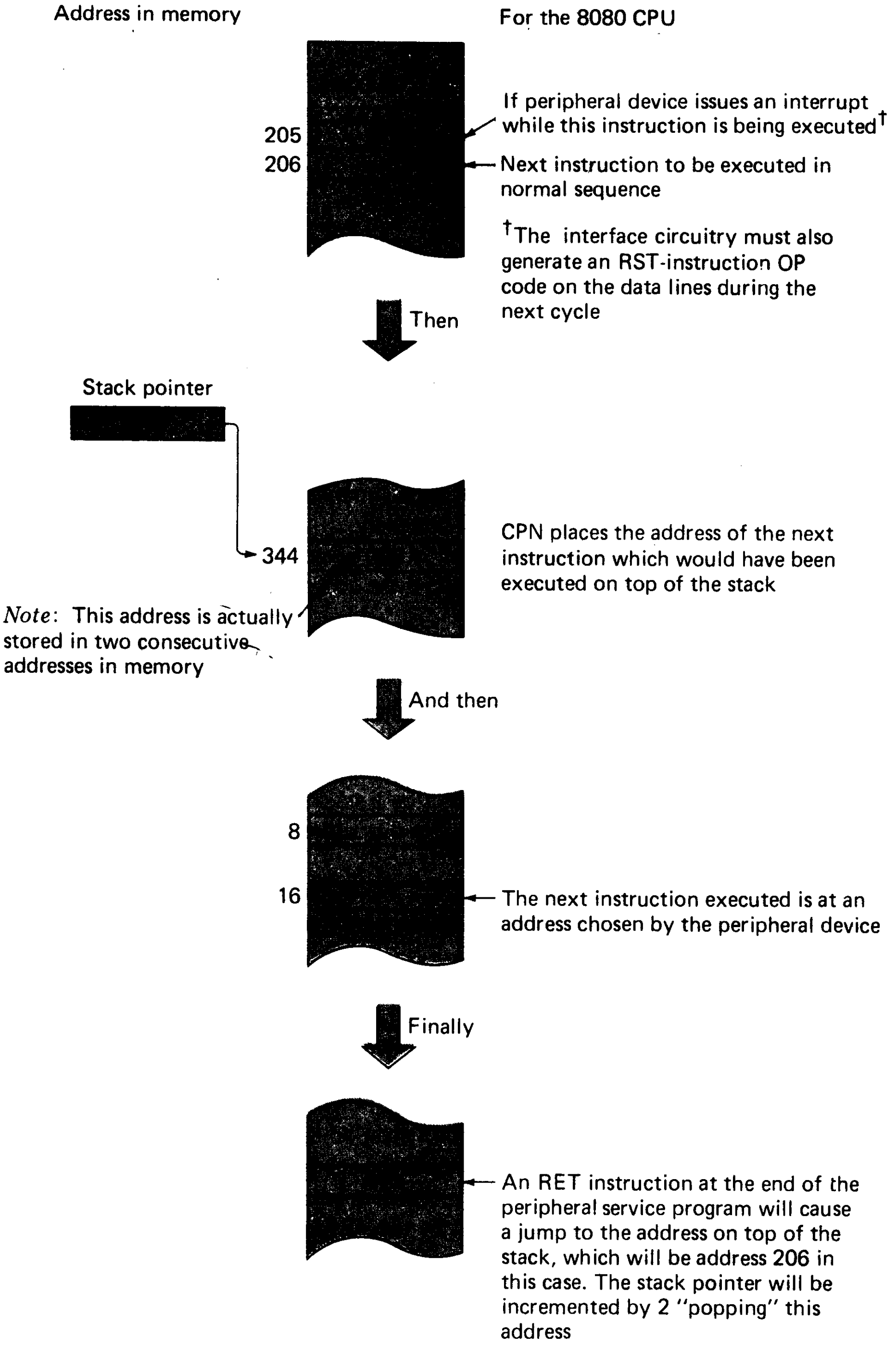
This computer employs what is now becoming the most used technique for subroutine calls and for servicing interrupts. For a subroutine jump a CALL instruction is used. The address of the subroutine is in the 16 bits (two memory addresses) following the CALL OP code. This instruction first increments the program counter to the address of the next instruction in sequence and then places (pushes) the contents of the program counter on a stack in the memory. The stack pointer (see Fig. 10.16) is adjusted to point to this address on the stack. The jump to the subroutine is then made.

At the end of the subroutine a RETURN instruction is used. This instruction specifies no address but simply causes a return to the address currently on top of the stack and then pops this address.

An investigation of this scheme will show that if a subroutine calls another subroutine which calls another subroutine which then calls the first subroutine, the successive addresses needed are stacked one on the other, and the subprograms will finally work their way back to the original calling program without loss of any of the necessary address links.

In preparing subroutines, some method must be agreed on for ''passing parameters'' into the subroutine. For example, if the subroutine's function is to find the square root of a number, then the original number must be passed to the subroutine, and the square root calculated by the subroutine must be passed back to the calling program. In this case the accumulator is a logical place to use for passing the number involved. Then each time the subroutine is to be used, the number whose square root is to be formed will be placed in the accumulator, the subroutine will be called, and at the end of the subroutine the square root will be in the accumulator.

In this computer interrupts are handled in a way similar to subroutine calls. The program being executed is interrupted, and the address of the next instruction which was to be executed is placed on top of the stack maintained by the stack



For the 8080 CPU

If peripheral device issues an interrupt while this instruction is being executed†

Next instruction to be executed in normal sequence

†The interface circuitry must also generate an RST-instruction OP code on the data lines during the next cycle

CPN places the address of the next instruction which would have been executed on top of the stack

Note: This address is actually stored in two consecutive addresses in memory

The next instruction executed is at an address chosen by the peripheral device

An RET instruction at the end of the peripheral service program will cause a jump to the address on top of the stack, which will be address 206 in this case. The stack pointer will be incremented by 2 "popping" this address

FIGURE 10.16

Interrupt servicing using a stack.

pointer. The interrupting device places an RST instruction on the data lines of the bus, and the 8080 next performs this instruction. The address section (NNN in Table 10.6) of this instruction contains the address in memory where the subroutine to service the interrupt is located. Returns from the interrupt servicing subroutine can then use the RETURN instruction to return to the original program. This is shown in Fig. 10.16.

When this kind of stacking of subprogram and interrupt addresses is used, it is possible to have subroutines interrupted, interrupts interrupted, and so on, and still return to each strip of instructions correctly as long as the stack does not overflow the area in memory allocated for it.

It is also necessary to save the registers in the CPU when interrupts occur if they are changed by the interrupt servicing subroutine. (The same applies for subroutine calls.) The interrupt program must take care of this saving and restoring of registers. Some idea of how this is done by using a stack can be gathered by examining the PUSH and POP instructions, and the Questions treat this in more detail.

We now examine a program using a subroutine call. The subroutine is to search a table for a specific character. If the character is found, the position of the character in the table is to be passed back to the calling routine.

Examination of the problem indicates that the following information must be passed to the subroutine: the location of the table in the memory, the length of the table (number of characters in the table), and the character to be searched for. To pass these three parameters, we choose registers *H* and *L* to point to the ending location of the table, put the number of characters into register *B*, and place the character to be searched for in the accumulator. The subroutine is then entered; its job is to find the character, place its position in the table in register *B*, and then return to the calling program. If the character is not in the table, *B* is made a 0.

A subroutine to perform this function is shown in Table 10.10. When the subroutine is entered, the register pair *H, L* points to the table's last location in memory, *B* gives the number of characters, and *A* contains the character to be located.

The name of the subroutine is SRCH. The ORG 30H statement, which occurs first, is an *assembler directive* which tells the assembler to "locate this subroutine beginning at location 30_{16} in memory."

The label SRCH identifies the subroutine. The CMP M instruction compares the last entry in the table (which is pointed to by the *H, L* pair) with the accu-

TABLE 10.10 A SEARCH SUBROUTINE

LABEL	OP CODE	OPERAND	COMMENTS
	ORG	30H	
SRCH	CMP	M	; IS CHAR = TABLE ENTRY?
	JZ	FINIS	; YES
	DCX	H	; GO TO NEXT ENTRY
	DCR	B	; DECREMENT B
	JNZ	SRCH	; IS SEARCH OVER?
FINIS	RET		; RETURN

TABLE 10.11 CALLING PROGRAM FOR SEARCH SUBROUTINE

LABEL	OP CODE	OPERAND	COMMENTS
	LXI	H, TBEND	; LOAD TABLE END ADDRESS
	MVI	B, 20H	; LOAD NO. OF CHARS
	LDA	CHAR	; LOAD CHARACTER
	CALL	SRCH	; CALL SUBROUTINE
	LDR	40H	; RETURN IS TO HERE
	...	...	

mulator. If they are equal, the Z status flag will be set, and the JZ instruction will cause a jump to FINIS. If they are not equal, the DCX H instruction subtracts 1 from the H, L pair, and then the DCR B subtracts 1 from B. When the DCR B instruction is executed, if B becomes 0, the Z flag will be set. The JNZ instruction tests this, and if another location is to be checked, it jumps back to SRCH; otherwise, the subroutine ends.

A possible calling sequence is shown in Table 10.11. The location of the end of the table is at TBEND in the memory, and the number of items is 20_{16}. These are loaded into the H, L pair and B, and the character to be searched for, here called CHAR, is loaded into the accumulator. Finally the subroutine is entered, by using a CALL SRCH instruction.

When the RET instruction in the subroutine is executed, the return will be to the LDR 40H instruction, which would be executed next. (This statement is placed in the program only for completeness and does not affect any operation discussed.)

6800 MICROPROCESSOR

10.12 Another widely used example of a microcomputer is the 6800 microprocessor-microcomputer system first developed by Motorola. (Chips for this system are also available from a number of other manufacturers.) The basic CPU is on a single 40-pin IC chip, as shown in Fig. 10.17. The 6800 has an 8-bit data bus and a 16-bit address bus (see Fig. 10.17). From a programming viewpoint, the CPU chip contains six basic registers, shown in Fig. 10.18.

1 *Accumulator A* This is an 8-bit accumulator.

2 *Accumulator B* This is an 8-bit accumulator.

3 *Index register* This is a single 16-bit index register.

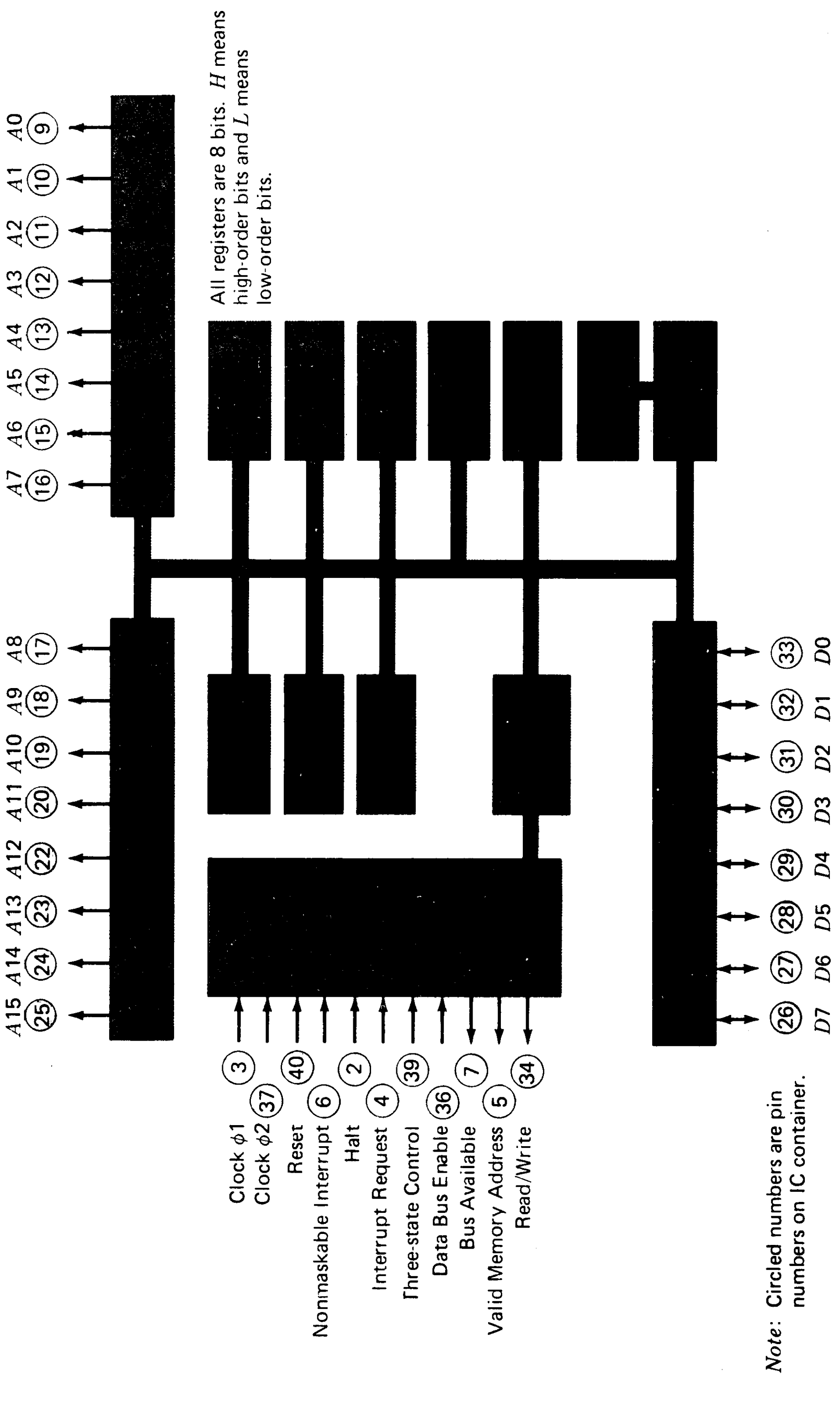
4 *Stack pointer* This is a 16-bit register which points to a stack in memory.

5 *Program counter* This is the instruction counter or program counter and contains 16 bits.

6 *Status register* This is a 6-bit register containing six flip-flops, H, I, N, Z, V, and C. The results of arithmetic and other operations are stored in these bits.

The instruction repertoire for this CPU chip includes over 100 different instructions. The operation code is 8 bits, the size of a word in memory. There are

6800 MICROPROCESSOR



All registers are 8 bits. H means high-order bits and L means low-order bits.

Note: Circled numbers are pin numbers on IC container.

FIGURE 10.17

Block diagram of CPU chip for 6800 microprocessor. (*Motorola Corp.*)

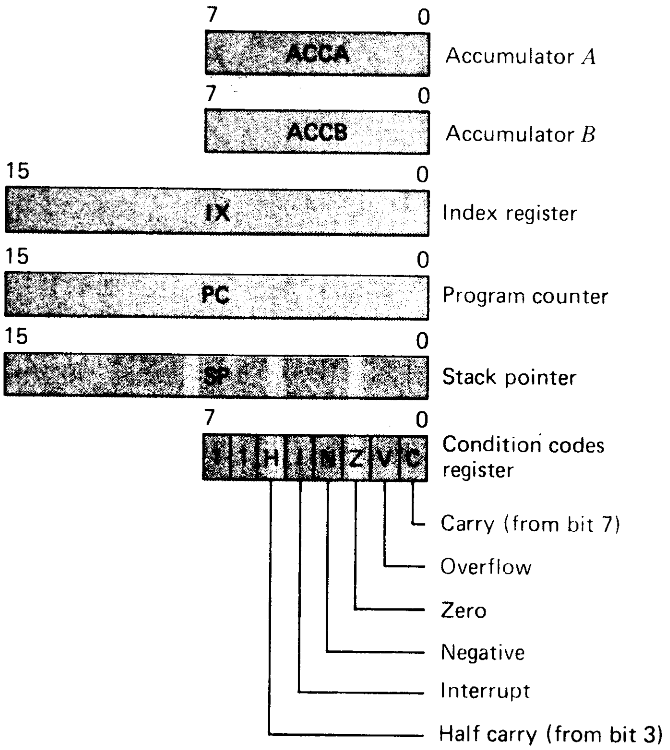


FIGURE 10.18

Basic registers used in programming the 6800.

seven different addressing modes, which are described in Table 10.12. A list of the instructions for this microprocessor chip is shown in Tables 10.13 to 10.15.

The way that conditional branch instructions operate deserves mention. When an arithmetic or boolean operation is performed, the status bits are set according to the result of this operation. Tables 10.16 and 10.17 show the status bits and detail their function. The BRANCH instructions use the values of these bits to determine whether a branch is to be made.

For instance, let us assume accumulator *A* is added to accumulator *B*. Then if the sum is negative, the *N* bit will be set to a 1. We also assume no overflow, so *V* will be set to a 0. Now if a BLT (branch if less than 0) instruction follows, the computer will branch to the address given in the address part of the instruction. If the result of the addition had been 0 or positive, no branch would have occurred, and the next instruction in sequence would be taken.

The *interrupt mask bit (I)* in the status register is set on when external input-output devices are allowed to interrupt the computer. When an interrupt occurs, the computer jumps to an interrupt servicing routine (program) which is stored in the memory. To simplify and shorten the interrupt servicing program, this microprocessor automatically transfers the values in all the CPU registers into a stack in the memory and places the address of these stored register contents in the stack pointer. The interrupt servicing program can then simply service the printer, reader, or whatever generated the interrupt and can later return the contents of the CPU

TABLE 10.12 ADDRESSING MODES FOR MICROPROCESSOR

MODE	OPERATION
Accumulator (ACCX) addressing	In accumulator-only addressing, either accumulator *A* or accumulator *B* is specified. These are 1-byte instructions.
Immediate addressing	In immediate addressing, the operand is contained in the second byte of instruction, except LDS and LDX, which have operands in second and third bytes of instruction. These are 2- or 3-byte instructions.
Direct addressing	In direct addressing, the address of the operand is contained in second byte of instructior. Direct addressing allows user to directly address lowest 256 bytes in machine, i.e., locations 0 through 255. Enhanced execution times are achieved by storing data in these locations. In most configurations it would be a random-access memory. These are 2-byte instructions.
Extended addressing	In extended addressing, the address contained in second byte of instruction is used as higher 8 bits of address of the operand. Third byte of instruction is used as lower 8 bits of address for the operand. This is an absolute address in memory. These are 3-byte instructions.
Indexed addressing	In indexed addressing, the address contained in second byte of instruction is added to index register's lowest 8 bits. Any carry generated is then added to higher-order 8 bits of index register. This result is then used to address memory. The modified address is held in a temporary address register so there is no change to index register. These are 2-byte instructions.
Implied addressing	In the implied addressing mode, the OP code gives the address (i.e., stack pointer, index register, etc.). These are 1-byte instructions.
Relative addressing	In relative addressing, the address contained in second byte of instruction is added to program counter's lowest 8 bits plus 2. The carry or borrow is then added to high 8 bits. This allows user to address data within a range of −125 to +129 bytes of present instruction. These are 2-byte instructions.

registers to their status when interrupted and restart the program where it was interrupted.

Maintaining these stored registers in a stack also enables the interrupt servicing program to be interrupted, since the contents of the registers are again placed in the stack. In this way several interrupts can follow one another, and the program can still service each interrupt in turn and then return to the original program, which was operating when the first interrupt occurred.

Details of the operating features of this microprocessor can be found in the manufacturers' manuals listed in the Bibliography.

Table 10.18 shows a short program for the 6800 microprocessor. Its purpose is to add a table of 8-bit bytes located in the memory starting at address 51_{16}. The number of bytes in the table is in location 50_{16}. The sum of the numbers is to be stored at location 0F in the memory. Carries from the addition are ignored.

TABLE 10.13 ACCUMULATOR AND MEMORY INSTRUCTIONS*

		ADDRESSING MODES					
		IMMED	DIRECT	INDEX	EXTND	IMPLIED	
OPERATIONS	MNEMONIC	OP	OP	OP	OP	OP	BOOLEAN ARITHMETIC OPERATION† (ALL REGISTER LABELS REFER TO CONTENTS)
Add	ADDA	8B	9B	AB	BB		A + M → A
	ADDB	CB	DB	EB	FB		B + M → B
Add accumulators	ABA					1B	A + B → A
Add with carry	ADCA	89	99	A9	B9		A + M + C → A
	ADCB	C9	D9	E9	F9		B + M + C → B
And	ANDA	84	94	A4	B4		A·M → A
	ANDB	C4	D4	E4	F4		B·M → B
Bit test	BITA	85	95	A5	B5		A·M
	BITB	C5	D5	E5	F5		B·M
Clear	CLR			6F	7F		00 → M
	CLRA					4F	00 → A
	CLRB					5F	00 → B
Compare	CMPA	81	91	A1	B1		A − M
	CMPB	C1	D1	E1	F1		B − M
Compare accumulators	CBA					11	A − B
Complement 1s	COM			63	73		$\overline{M}$ → M
	COMA					43	$\overline{A}$ → A
	COMB					53	$\overline{B}$ → B
Complement 2s (negate)	NEG			60	70		00 − M → M
	NEGA					40	00 − A → A
	NEGB					50	00 − B → B
Decimal adjust, *A*	DAA					19	Converts binary addition of BCD characters into BCD format
Decrement	DEC			6A	7A		M − 1 → M
	DECA					4A	A − 1 → A
	DECB					5A	B − 1 → B
Exclusive OR	EORA	88	98	A8	B8		A ⊕ M → A
	EORB	C8	D8	E8	F8		B ⊕ M → B
Increment	INC			6C	7C		M + 1 → M
	INCA					4C	A + 1 → A
	INCB					5C	B + 1 → B
Load accumulator	LDAA	86	96	A6	B6		M → A
	LDAB	C6	D6	E6	F6		M → B
Inclusive OR	ORAA	8A	9A	AA	BA		A + M → A
	ORAB	CA	DA	EA	FA		B + M → B